Avant-Propos

Présentation du document

This document is a re-lecture of the tutorial "Découvrez Ogre 3D". The original tutorial was written by Julien Chichignoud on Openclassroom (aka "Le site des zéros").

Writing this document was an attempt to make something like an active reading, and thus have a more attentive lecture and deepest understanding of Ogre's concepts. We modified a lot (sentences, structure, ...) but the essence is the same as the original tutorial.

We wish to highligh the fact that this document is a personal interpretation, as the implementation.

The original tutorial is in French and as we spent the most of our strength in the comprehension of Ogre we did not take time to translate the original tutorial in French.

Licence

The rest of the document will be now in French, sorry for those who cannot take advantage of the document.

Première partie

OGRE
Installation, Premier programme

Chapitre 1

Premier pas

1.1 Sources

J'ai d'abord voulu suivre le tutoriel du site du zéro `http://fr.openclassrooms.com/informatique/cours/decouvrez-ogre-3d/`

Mais ce tutoriel du site des zéros semble plus axé visual studio, par contre un lien est donné pour la compilation de projets sous Ubuntu `http://geenux.wordpress.com/2010/03/18/installation-de-ogre-1-7-et-compilation-avec-cmake-sous-ubuntu/`

1.2 Installation

Installation avec synaptic des packages suivants :

1. libogre-dev
2. ogre-samples
3. ogre-samples-data
4. libogre-1.7.4
5. libois-1.3.0
6. libois-dev

Chapitre 2

Premiere compilation

2.1 Premier programme

— Création d'un répertoire contenant les fichiers suivants :

```
Hraesvelg:~/Documents/workspace/3D/OGRE/ExampleApplication$ ls -l
total 24
-rw-rw-r-- 1 olivier olivier 1109 fvr. 23 12:16 CMakeLists.txt
-rw-rw-r-- 1 olivier olivier  844 fvr. 23 12:02 helloworld.cpp
-rw-rw-r-- 1 olivier olivier 4894 fvr. 23 12:18 Makefile
-rw-r--r-- 1 olivier olivier  446 fvr. 23 12:20 plugins.cfg
-rw-r--r-- 1 olivier olivier 1391 fvr. 23 12:20 resources.cfg
```

— helloworld.cpp :

```
#include ''ExampleApplication.h''

class TutorialApplication: public ExampleApplication
{
protected:
public:
    TutorialApplication()
    {
    }

    ~TutorialApplication()
    {
    }
protected:
    void createsc\'ene(void)
    {
    }
};

#if OGRE_PLATFORM == OGRE_PLATFORM_WIN32
#define WIN32_LEAN_AND_MEAN
#include ''windows.h''

INT WINAPI WinMain( HINSTANCE hInst, HINSTANCE, LPSTR strCmdLine,↩
     INT )
#else
int main(int argc, char **argv)
#endif
{
```

```
29      // Create application object
30      TutorialApplication app;
31
32      try {
33          app.go();
34      } catch( Exception& e ) {
35  #if OGRE_PLATFORM == OGRE_PLATFORM_WIN32
36          MessageBox( NULL, e.what(), ''An exception has occurred!''↩
                ', MB_OK | MB_ICONERROR | MB_TASKMODAL);
37  #else
38          fprintf(stderr, ''An exception has occurred: %s\n'',
39                  e.what());
40  #endif
41      }
42
43      return 0;
44  }
```

— CMakeLists.txt est une copie modifiée du CMakeLists.txt donnée dans le tutoriel de compilation sous Ubuntu. Les lignes commencant par # sont les lignes originales que j'ai du modifier.

```
1   project(helloworld)
2   cmake_minimum_required(VERSION 2.6)
3
4   \# \~ set(CMAKE_MODULE_PATH ''/usr/lib/OGRE/cmake/'')
5   set(CMAKE_MODULE_PATH ''/usr/share/OGRE/cmake/modules'')
6
7   #set(CMAKE_CXX_FLAGS ''-Wall -W -Werror -ansi -pedantic -g'')
8
9   # Il s sagit du tutoriel d exemple, qui utilise quelques fichiers↩
        predefini de Ogre. Il faut indiquer a cmake o\'{u} se ↩
        trouvent les includes en question
10  # \~ include_directories (''/usr/share/OGRE/Samples/Common/↩
        include/'')
11  include_directories (''/usr/share/OGRE-1.7.4/Samples/Common/↩
        include/'')
12
13  # Bien sur, pour compiler Ogre, il faut le chercher, et definir ↩
        le repertoire contenant les includes.
14  find_package(OGRE REQUIRED)
15  include_directories (${OGRE_INCLUDE_DIRS})
16
17  # L'exemple depend aussi de OIS, une lib pour gerer la souris, ↩
        clavier, joystick...
18  find_package(OIS REQUIRED)
19
20  # On definit les sources qu'on veut compiler
21  SET(SOURCES
22  helloworld.cpp)
23
24  # On les compile
25  add_executable (
26  helloworld ${SOURCES}
27  )
28
29  # Et pour finir, on lie l'executable avec les librairies que ↩
        find_package nous a gentillement trouve.
30  target_link_libraries(helloworld ${OGRE_LIBRARY} ${OIS_LIBRARY})
31  %
```

plugins.cfg et resources.cfg sont copiés de /usr/share/OGRE-1.7.4

2.1.1 Compilation

les commandes à faire sont :
— cmake.
— make
— ./helloworld

2.2 Premiere compilation sous CodeBlocks

A tenter!!!!
`http://fr.openclassrooms.com/informatique/cours/decouvrez-ogre-3d/configuration-d-un-projet-3`

2.3 Premiere compilation sous kDevelop

J'ai suivi
`http://www.ogre3d.org/tikiwiki/tiki-index.php?page=Setting+Up+An+Application+-+KDevelop+-+Linux`.
Je copie dans le répertoire ogretest le répertoire d'exemple Sample_Water. Le processus foire.

Chapitre 3

Code de base

3.1 Source

`http://fr.openclassrooms.com/informatique/cours/decouvrez-ogre-3d/le-code-de-base-1`

Ce code de base est en fait le même que celui fait dans le chapitre Premiere Compilation, sauf qu'alors on avait que un seul fichier cpp, on rajoute içi un fichier.h qui est inclus dans le fichier main.cpp.

Deuxième partie

OGRE
Insérer des objets dans la scène

3.2 les entités

3.2.1 Une entité c'est quoi

Une entité est l'ensemble des informations attachées aux polygones constituant un objet 3D.
Un Mesh étant l'ensemble des polygones constituant un objet 3D, une entité est l'ensemble des informations attachées à un Mesh :

— les vertices (sommets des polygones),
— les textures,
— un squelette si le modele est sujet à des animations
— ...

Tous les objets solides qui apparaissent à l'écran sont donc des entités et sont représentés par une seule et même classe dans Ogre : la classe Entity.

3.2.2 Le sceneManager

C est au sceneManager que revient la tâche de créer tous les objets que peut contenir la scène (tous les modèles, lumières, caméra et autres objets) et de nous permettre ensuite d'y accéder.
Par conséquent, l'insertion d'objets dans la scène passe toujours par lui (ou par l'un des éléments déjà insérés) et par les méthodes qu'il propose pour cela. Il en existe diverses variantes en fonction de la scène que l'on veut réaliser ; selon que celle-ci sera par exemple en intérieur ou en extérieur, par exemple.

Toutes les applications Ogre doivent donc avoir un sceneManager pour pouvoir fonctionner, puisque c'est lui qui s'occupe de tout ! La classe ExampleApplication ne fait pas exception et possède donc un attribut msceneMgr, qui est un pointeur sur le sceneManager de l'application et qui nous permettra dans les parties suivantes d'agrémenter notre scène avec des objets.

3.2.3 Créer une entité

L'entité doit être créée par le sceneManager pour pouvoir être ajoutée à la scène. La méthode createEntity() permet de faire cela.

```
1 Entity *head= msceneMgr->createEntity(''Tete'',''ogrehead.mesh'')↩
    ;
```

1

— Le premier paramètre est le nom que vous souhaitez donner au mesh. Ce nom doit être unique !
— Le second paramètre est le nom du fichier que vous voulez charger. Notez l'extension.mesh, qui est le format de fichiers pour les modèles reconnu par Ogre.

Le fichier spécifié (avec le deuxième paramètre) se trouve, accompagné d'autres modèles d'exemples, dans le dossier OgreSDK/media/models, qui doit être correctement renseigné dans le fichier resources.cfg pour qu'Ogre puisse le trouver lors de l'exécution. [2]

1. Pourquoi utilise t on -¿ pour appeler une méthode d'une classe ?
2. Mais à quoi sert ce fichier .mesh ?

Le mesh a été ajouté à la scène mais il nous manque encore une chose avant de pouvoir l'afficher à l'écran...

3.3 Les Noeuds de la scène

3.3.1 L'utilité des Noeuds

Dans Ogre, lorsque l'on souhaite manipuler une entité (un personnage, une lumière, une caméra...), les déplacements que l'on veut effectuer se font par l'intermédiaire d'un noeud de scène, ou sceneNode.

Un sceneNode est un objet invisible auquel on va pouvoir attacher un nombre indéfini d'entités, lesquelles deviennent solidaires de ce noeud et subissent donc les même transformations que lui. C'est donc une sorte de conteneur qui contient les informations de positionnement de chacune des entités de la scène qui lui sont rattachées.

Bien sûr on pourrait déplacer nos entités directement mais avec un noeud on pourra déplacer en une fois toutes les entités attachées au noeud.

Quoi qu'il en soit, attacher chaque entité à un noeud est primordial, sans quoi elle ne s'affichera pas dans votre scène !

3.3.2 Créer un noeud

Pour créer un noeud de scène on devra passer par un noeud déjà existant, ce qui va nous permettre d'avoir des relations d'héritage entre nos noeuds.

On utilisera une méthode dédiée :

```
1 sceneNode *noeudEnfant = noeudParent->createChildsceneNode(''↩
    enfant'', Vector3::ZERO, Quaternion::IDENTITY);
```

Mais comment fait-on pour le premier noeud qu'on va créer ? Le noeud "racine" existe dès que le sceneManager est créé, c'est un noeud comme un autre avec les même [3] méthodes mais ce noeud est unique.

Nous récupérons ce noeud racine de la scène à l'aide de l'instance du sceneManager :

```
1 sceneNode *node= msceneMgr->getRootsceneNode()->↩
    createChildsceneNode(''nodeTete'', Vector3::ZERO, Quaternion↩
    ::IDENTITY);
```

La méthode getRootsceneNode() nous permet de récupérer un pointeur sur le noeud racine unique de la scène. On appelle ensuite sa méthode createChildsceneNode pour lui ajouter un nouveau noeud fils.

Notez qu'aucun des paramètres de la méthode n'est obligatoire, pour information :

- Le premier argument est le nom que vous voulez donner à votre noeud. Sur le même principe que les entités ce nom pourra être utilisé pour récupérer un pointeur vers le noeud en question.
- Le deuxième argument est la position initiale du noeud.

3. "les même" faut il un "s" à "même" ?

— Le troisième argument est le quaternion avec lequel vous voulez initialiser votre noeud.

Sachez pour le moment qu'un quaternion est un objet mathématique qui permet de faire faire aux objets des rotations dans l'espace [4]. Quaternion : :IDENTITY [5]lui dit de ne pas faire de rotation.

Avec ce code en main, vous pouvez simplement attacher l'entité précédemment créée au noeud avec la ligne suivante :

```
1 node->attachObject(head);
```

En compilant, vous devriez voir la tête d'un ogre au milieu de l'écran. Vous pouvez déplacer la caméra avec Z, S, Q, D et la souris pour voir ce que ça donne de plus près.

3.3.3 Code

PremiereApplication.cpp

Listing 3.1 – PremiereApplication.cpp : Instanciation d'entité

```
1  #include "PremiereApplication.h"
2
3  void PremiereApplication::createScene()
4  {
5      //creation d une entite
6      Entity *head= mSceneMgr->createEntity("Tete", "ogrehead.mesh"↩
           );
7
8      //creation d un noeud
9      SceneNode *node= mSceneMgr->getRootSceneNode( )->↩
           createChildSceneNode( "nodeTete " , Vector3::ZERO, ↩
           Quaternion::IDENTITY);
10
11     node->yaw(Radian(Math::PI));
12     node->yaw(Radian(Math::PI));
13
14     Vector3 position = Vector3(30.0, 50.0, 0.0);
15     node->setPosition(position);
16
17     node->setPosition(30.0, 50.0, 0.0);
18     node->translate(-30.0, 50.0, 0.0);
19
20
21     //attachement de l entite au noeud
22     node->attachObject ( head );
23 }
```

3.4 Créer un mesh

Nous pouvons rajouter un sol à notre scène, pour cela nous allons créer nous-mêmes [6] un nouveau mesh. Étant donné que nous n'avons besoin que d'un

4. Que des rotations ?
5. comment écrire Quaternion : :IDENTITY de maniÃ¨re élégante ?
6. il faut un "s" à nous-mêmes ?

plan pour le sol, le mesh peut très simplement être créer dans le code de notre application.

3.4.1 Le mesh

Il existe une classe Plane [7] qui va nous permettre de générer... un plan qui représentera le sol. [8]

Pour créer un plan nous appelons Plane avec les paramètres suivants :
- le premier paramètre permet de définir le vecteur normal au plan à créer (ici l'axe Y pour que notre plan soit horizontal).
- le second paramètre est la distance à l'origine de la scène dans le sens du vecteur normal (ici, je mets 0 pour que mon mesh plan soit centré).

```
1        Plane plan(Vector3::UNIT_Y, 0);
```

Une fois le plan créé, il faut que l'on crée un mesh, c'est-à-dire l'objet 3D en lui-même (la représentation du plan) qui sera visible dans la scène. Pour cela, on utilise le Mesh Manager, qui va s'occuper de créer les faces de notre mesh.

```
1 MeshManager::getSingleton().createPlane(''sol'', ↩
    ResourceGroupManager::DEFAULT_RESOURCE_GROUP_NAME, plan, 500,↩
     500, 1, 1, true, 1, 1, 1, Vector3::UNIT_Z);
```

Quelques explications sur cette ligne s'imposent.
- Tout d'abord, la méthode statique getSingleton() permet de récupérer un objet instancié de façon unique, donc ici notre MeshManager.
- Les deux premiers paramètres correspondent respectivement au nom que l'on veut donner à notre mesh et au nom du groupe auquel on veut qu'il appartienne.
- Suivent ensuite le nom du plan à modéliser, puis la largeur et la hauteur qu'il doit avoir, puis le nombre de subdivisions du plan dans ces deux sens. Plus il y a de subdivisions, plus il y a de polygones dans notre mesh.
- Le booléen suivant indique que les normales sont perpendiculaires au plan.
- Les trois paramètres suivants sont le nombre de textures que l'on va pouvoir assigner au plan, puis le nombre de fois que la texture sera répétée dans les deux directions.
- le dernier paramètre est le vecteur indiquant la direction du haut du mesh. Attention : il ne faut pas le confondre avec la normale du plan, qui est différente.

Il reste encore des paramètres par défaut que l'on verra plus tard.

Enfin, nous allons revenir vers un code connu : nous allons créer l'entité qui représentera le plan. C'est le même principe que tout à l'heure :
- tout d'abord, on crée une entité à partir du scène Manager en la nommant et en lui indiquant le mesh à utiliser.

7. de même que pour quaternion.entity marquer les nom de classe de maniÃ¨re élégante pour les discerner du texte serait bien

8. En plus de la classe Plane, vous trouverez aussi des classes Box (pour les cubes) et Sphere qui fonctionnent sur le même principe.

— On crée ensuite un nouveau noeud à partir du noeud racine et on l'attache à notre entité.

```
//creation d'une entite
Entity *ent= msceneMgr->createEntity(''EntiteSol'', ''sol'');

//creation d'un nouveau noeud
node = msceneMgr->getRootsceneNode()->createChildsceneNode();

//on attache le noeud a notre entite
node->attachObject(ent);
```

3.4.2 Le matériau

Nous allons finir en ajoutant une texture au sol : de l'herbe. Pour cela, il suffit de rajouter la ligne suivante après la création de l'entité :

```
ent->setMaterialName(''Examples/GrassFloor'');
```

Si vous voulez connaître les matériaux fournis avec Ogre, il vous suffit d'aller dans le dossier media/materials/scripts. Ici, on prend le matériau GrassFloor enregistré dans le fichier Examples.material.

Les textures correspondantes se trouvent dans le dossier media/materials/-textures, si vous voulez faire des essais.

Vous pouvez maintenant exécuter votre programme.

Lancez l'application et remontez la caméra avec la souris et les touches de déplacement, vous devriez voir quelque chose ressemblant à la capture suivante.

Image utilisateur

Euh... La tête d'Ogre est coupée par le sol en herbe...

En effet, notre plan est centré sur l'origine de la scène, et l'on a aussi placé notre tête à l'origine. Mais quelle partie de la tête est à l'altitude 0 ?

Ici, c'est donc un point au milieu de la tête, puisque le plan d'herbe passe par là. Cependant, ce point n'est pas nécessairement au milieu de l'objet que vous intégrez. Cela dépend de la personne qui a modélisé l'objet et qui a donc décidé par rapport à quel point on allait définir la position du mesh. Pour un personnage, on pourrait mettre ce point à ses pieds, pour que l'altitude 0 corresponde effectivement au moment où le personnage touche le sol avec ses pieds.

Pour corriger cela, il va falloir remonter notre noeud lié à notre entité. C'est l'objet du prochain chapitre.

3.4.3 Code

PremiereApplication.cpp

Listing 3.2 – PremiereApplication.cpp : Création d'un sol

```
#include "PremiereApplication.h"

void PremiereApplication::createScene()
{
    //creation d une entite
    Entity *head= mSceneMgr->createEntity("Tete", "ogrehead.mesh"↩
        );
```

```
7
8       //creation d un noeud
9       SceneNode *node= mSceneMgr->getRootSceneNode( )->↩
            createChildSceneNode( "nodeTete " , Vector3::ZERO, ↩
            Quaternion::IDENTITY);
10
11      node->yaw(Radian(Math::PI));
12      node->yaw(Radian(Math::PI));
13
14      //setPosition place le noeud aux coord passees en parametres
15      Vector3 position = Vector3(30.0, 50.0, 0.0);
16      node->setPosition(position);
17
18      node->setPosition(30.0, 50.0, 0.0);
19      /*equivalent a
20      Vector3 position = Vector3(30.0, 50.0, 0.0);
21      node->setPosition(position);
22      */
23
24      //deplace le noeud par rapport a sa position actuelle
25      node->translate(-30.0, 50.0, 0.0); //par defaut la trnslt se ↩
            fait par rap a TS_WORLD
26
27      //attachement de l entite au noeud
28      node->attachObject ( head );
29
30      //creation d un plan
31      Plane plan(Vector3::UNIT_Y, 0);
32
33      //creation d un mesh cad l objet 3d visible ds la scene
34      MeshManager::getSingleton().createPlane("sol",
35                  ResourceGroupManager::DEFAULT_RESOURCE_GROUP_NAME↩
                        ,
36                  plan, 500, 500, 1, 1, true, 1, 1, 1, Vector3::↩
                        UNIT_Z);
37
38      //entite qui representera le plan
39      Entity *ent= mSceneMgr->createEntity("EntiteSol", "sol");
40
41      //ajout du materiau a l entite
42      ent->setMaterialName("Examples/GrassFloor");//texture de ↩
            pelouse
43      /*les differents materiaux sont sous /media/materials/scritps↩
            , par ex:
44      ent->setMaterialName("Examples/WaterStream");//texture d eau ↩
            animee*/
45
46      //creation d un noeud
47      node = mSceneMgr->getRootSceneNode()->createChildSceneNode();
48      node->attachObject(ent);
49  }
```

A cause de la position de la caméra, il se peut alors que le sol ne soit pas visible.

Troisième partie

Se repérer dans l'espace

3.5 Le système de coordonnées

3.5.1 Le repère "main droite"

Le repère est orthogonal et orthonormé (les axes/les vecteurs directeurs de ces axes sont perpendiculaires les uns aux autres et de longueur 1)

3.5.2 Pourquoi main droite ?

Regardez votre main droite.
— Votre pouce représente l'axe X,
— votre index représente l'axe Y,
— votre majeur représente l'axe Z.

La direction dans laquelle pointe chacun de vos doigts définit le sens de chaque axe.

Image utilisateur cf(`fr.openclassrooms.com/informatique/cours/decouvrez-ogre-3d/le-systeme-de-coordonnees-1`)

3.5.3 Repère local, repère absolu

Pour l'instant, nous n'avons toujours pas défini comment le repère était placé dans la scène. C'est une question de convention adoptée pour les applications 3D pour le repère de la scène :
— l'axe Y dirigé vers le haut
— les axes X et Z dans un plan horizontal.

Seulement [9], la scène n'est pas la seule à avoir son repère. En effet, chaque objet possède son propre repère appelé repère local. Lorsque l'objet se déplace ou tourne sur lui-même, le repère local fait de même. L'orientation du repère local est la suivante :
— l'axe Y est dirigé vers le haut de l'objet, pour la scène,
— l'axe X est dirigé vers sa droite
— l'axe Z vers l'arrière de l'objet.

Ci-dessous [10], j'ai représenté en noir le repère de la scène, et en bleu le repère local de la voiture, en respectant la convention que j'ai donnée. Image utilisateur cf(`fr.openclassrooms.com/informatique/cours/decouvrez-ogre-3d/le-systeme-de-coordonnees-1`)

Nous verrons à quoi servent ces différents repères lorsque l'on commencera à déplacer nos objets.

3.5.4 Yaw, pitch, roll

yaw pitch roll sont les désignations anglaises pour les rotations autour des axes Y, X et Z respectivement. On peut traduire ces termes par lacet (yaw), tangage (pitch) et roulis (roll), qui sont utilisés par exemple en aéronautique ou en navigation.

Ces trois termes se retrouveront dans les noms qu'Ogre donne aux méthodes permettant d'effectuer des rotations.

9. comment faire pour sauter une ligne avant le "Seulement" ?
10. comment faire pour sauter une ligne avant le "Ci-dessous" ?

Voici tout de suite un schéma illustrant les rotations qui s'appliquent à chaque axe :

Image utilisateur cf(`fr.openclassrooms.com/informatique/cours/decouvrez-ogre-3d/le-systeme-de-coordonnees-1`)

Tout comme l'orientation des axes de notre repère, il y a un sens direct et un sens indirect pour les rotations ! Vous tournerez dans le sens direct lors d'une rotation contraire [11] au sens des aiguilles d'une montre.

Pour effectuer une rotation, on appelle la méthode correspondante pour le noeud :

```
1        node->yaw(Radian(Math::PI));
```

Ceci fera faire un demi-tour au noeud par rapport à son axe vertical tandis que les méthodes pitch() et roll() s'utilisent de façon analogue pour les autres axes.

La rotation se fait par défaut par rapport au repère local. Il faut renseigner le second paramètre si vous voulez qu'il en soit autrement (voir la section suivante).

Les angles doivent être entrés en radians. Pour utiliser tout de même des degrés dans Ogre, vous devrez utiliser la classe Degree. La ligne de code précédente est équivalente à ceci :

```
1        node->yaw(Degree(180));
```

3.6 Déplacer des objets

3.6.1 Bouger un noeud de scène

Nous allons maintenant déplacer notre tête d'ogre pour vérifier la théorie et enfin sortir notre tête de terre !

Pour cela, nous allons donc passer par le noeud auquel est rattaché notre mesh. Celui-ci possède deux méthodes qui peuvent nous servir.

3.6.2 setPosition()

La méthode setPosition() prend en paramètres les trois coordonnées X, Y et Z du point auquel on désire placer le noeud. On peut aussi lui passer un Vector3, qui contiendra lui-même ces coordonnées.

Les deux codes suivants sont donc équivalents.

```
1        Vector3 position = Vector3(30.0, 50.0, 0.0);
2        node->setPosition(position);
```

ou

```
1        node->setPosition(30.0, 50.0, 0.0);
```

La tête s'est maintenant déplacée vers la droite de 30 unités et de 50 unités vers le haut.

11. "rotation contraire" ¡-¿ "sens direct", est ce correct ?

3.6.3 translate()

La méthode translate() déplace le noeud par rapport à sa position actuelle plutôt que par rapport à l'origine de la scène.

Elle prend les mêmes paramètres que la méthode setPosition(), mais avec un paramètre supplémentaire, défini par défaut, indiquant le noeud par rapport auquel on va se déplacer.

```
1        node->translate(-30.0, 50.0, 0.0);
```

En ajoutant cette ligne après la précédente, notre objet se retrouve donc maintenant à la position (0, 100, 0) dans la scène, ce qui est suffisant pour qu'il surplombe son petit jardin.

Image utilisateur

Le paramètre supplémentaire (par rapport à setPosition) permet de définir par rapport à quel repère on va déplacer le noeud.

Les trois valeurs possibles sont :

```
1    Node::TS_LOCAL //va deplacer le noeud par rapport au repere ↩
         local
2    Node::TS_PARENT //va deplacer le noeud au repere du noeud ↩
         parent
3    Node::TS_WORLD //va deplacer le noeud au repere de la scene, ↩
         qui est le repere absolu.
```

12

3.6.4 Concrètement, ça veut dire quoi ?

Tout à l'heure, lorsque l'on a effectué une translation, on l'a fait par défaut par rapport au repère TS_WORLD, c'est-à-dire avec les axes tels que je vous les ai présentés précédemment. Maintenant, nous pouvons déplacer notre noeud par rapport au repère local de son noeud père par exemple, ou bien même par rapport à son propre repère local.

Mais à quoi ça sert de s'embêter avec ces paramètres ? On risque de faire des erreurs si l'on se place par rapport à un repère différent de la scène !

Prenons un exemple. Vous avez un vaisseau spatial qui peut se trouver dans n'importe quelles position et orientation de l'espace. Comment savoir facilement dans quelle direction je dois faire ma translation pour qu'on le voit aller en avant ?

Réponse : je n'ai pas à m'en occuper ! En effet, l'axe qui va de l'avant vers l'arrière du vaisseau est l'axe Z, dans son repère local. Par conséquent, je n'ai qu'à dire à mon vaisseau d'avancer le long de l'axe Z (dans le sens négatif pour aller à l'avant) par rapport à son repère local. Et Ogre s'occupera gentiment de faire les calculs pour placer mon vaisseau correctement dans la scène.

Image utilisateur `http://fr.openclassrooms.com/informatique/cours/decouvrez-ogre-3d/deplacer-des-objets`

Si vous avez compris cela, le paramètre TS_PARENT devrait suivre tout seul. Reprenons notre engin spatial. Sur ce vaisseau, on trouve R2D2 en train de se déplacer vers la droite, correspondant donc à l'axe X local du noeud du vaisseau. Pour effectuer cette translation, je n'ai qu'à demander à Ogre de déplacer mon

12. Pour prÃ©senter les 3 valeurs possibles ne devrais je pas faire une liste itemize plutot que d'utiliser un bloc de code ?

robot le long de l'axe des abscisses par rapport au noeud du vaisseau (qui serait logiquement le noeud parent).

Vous commencez à comprendre l'intérêt des relations de parenté entre les noeuds ?

3.7 La caméra

La caméra, c'est l'élément qui définit la position de notre point de vue dans la scène, dans quelle direction on regarde, mais aussi jusqu'à quelle distance il est possible de voir s'afficher les objets éloignés.

Comme tous les éléments de base, un attribut caméra est présent dans la classe ExampleApplication. Sans elle nous n'aurions pas encore pu voir notre scène, vu que nous n'avons rien fait pour la créer !

3.7.1 Création

La caméra est créée par la méthode createCamera() de la classe ExampleApplication, nous allons tout de suite redéfinir cette méthode pour partir sur des bases connues. L'attribut correspondant à la caméra est appelé mCamera, nous pouvons donc l'utiliser pour créer notre caméra.

Comme c'est un objet qui se trouve dans la scène, nous allons passer par le sceneManager. Comme pour les noeuds ou les entités, vous pourrez donner un nom à votre caméra sous forme d'une chaîne de caractères.

Ajoutez la méthode createCamera() à votre classe PremiereApplication et ajoutez-y la ligne suivante.

```
1        mCamera = msceneMgr->createCamera(''Ma Camera'');
```

3.7.2 Placement

Maintenant, il va nous falloir placer la caméra et l'orienter. Le placement se fait avec la méthode setPosition().

La seconde méthode utilisée s'appelle lookAt() et, comme son nom l'indique, elle permet de déterminer le point de la scène que regarde notre caméra. On lui fournit un Vector3 ou bien trois réels correspondant aux coordonnées désirées.

```
1        //placement de la camera
2        mCamera->setPosition(Vector3(-100.0, 150.0, 200.0));
3        //point de la scene que regarde notre camera
4        mCamera->lookAt(Vector3(0.0, 100.0, 0.0));
```

Enfin, on peut aussi indiquer les distances near clip et far clip, qui sont les distances minimale et maximale [13] auxquelles doit se trouver un objet pour être affiché à l'écran.

```
1        mCamera->setNearClipDistance(1);
2        mCamera->setFarClipDistance(1000);
```

13. "les distances minimale et maximale" il faut pas de "s" à "max/minimale""

3.7.3 Code

PremiereApplication.cpp

Listing 3.3 – PremiereApplication.cpp : Création de la caméra

```
1
2  #include "PremiereApplication.h"
3
4  void PremiereApplication::createScene()
5  {
6      //creation d une entite
7      Entity *head= mSceneMgr->createEntity("Tete", "ogrehead.mesh"↩
           );
8
9      //creation d un noeud
10     SceneNode *node= mSceneMgr->getRootSceneNode( )->↩
           createChildSceneNode( "nodeTete " , Vector3::ZERO, ↩
           Quaternion::IDENTITY);
11
12     node->yaw(Radian(Math::PI));
13     node->yaw(Radian(Math::PI));
14
15     //setPosition place le noeud aux coord passees en parametres
16     Vector3 position = Vector3(30.0, 50.0, 0.0);
17     node->setPosition(position);
18
19     node->setPosition(30.0, 50.0, 0.0);
20     /*equivalent a
21     Vector3 position = Vector3(30.0, 50.0, 0.0);
22     node->setPosition(position);
23     */
24
25     //deplace le noeud par rapport a sa position actuelle
26     node->translate(-30.0, 50.0, 0.0); //par defaut la trnslt se ↩
           fait par rap a TS_WORLD
27
28     //attachement de l entite au noeud
29     node->attachObject ( head );
30
31     //creation d un plan
32     Plane plan(Vector3::UNIT_Y, 0);
33
34     //creation d un mesh cad l objet 3d visible ds la scene
35     MeshManager::getSingleton().createPlane("sol",
36                 ResourceGroupManager::DEFAULT_RESOURCE_GROUP_NAME↩
                       ,
37                 plan, 500, 500, 1, 1, true, 1, 1, 1, Vector3::↩
                       UNIT_Z);
38
39     //entite qui representera le plan
40     Entity *ent= mSceneMgr->createEntity("EntiteSol", "sol");
41
42     //ajout du materiau a l entite
43     ent->setMaterialName("Examples/GrassFloor");//texture de ↩
           pelouse
44     /*les differents materiaux sont sous /media/materials/scritps↩
           , par ex:
45     ent->setMaterialName("Examples/WaterStream");//texture d eau ↩
           animee*/
46
47     //creation d un noeud
```

```
48      node = mSceneMgr->getRootSceneNode()->createChildSceneNode();
49      node->attachObject(ent);
50  }
51
52  /*definit la position de notre point de vue*/
53  void PremiereApplication::createCamera()
54  {
55      //creation de la camera
56      mCamera = mSceneMgr->createCamera("Ma Camera");
57
58      //position de la camera
59      mCamera->setPosition(Vector3(-100.0, 150.0, 200.0));
60
61      //permet de determiner le point de la scene que regarde notre↩
            camera
62      mCamera->lookAt(Vector3(0.0, 100.0, 0.0));
63
64      //definition des distances de near clip et de far clip, qui
65      //sont les distances minimale et maximale auxquelles doit se
66      //trouver un objet pour Ãªtre affichr Ã  l'Ã©cran.
67      mCamera->setNearClipDistance(1);
68      mCamera->setFarClipDistance(1000);
69  }
```

PremiereApplication.h

Listing 3.4 – PremiereApplication.h : Création de la caméra

```
1   using namespace std;
2
3   #include <ExampleApplication.h>
4
5   class PremiereApplication : public ExampleApplication
6   {
7   public:
8       void createScene();
9       void createCamera();
10  };
```

main.cpp

Listing 3.5 – main.cpp : Création de la caméra

```
1   #include <Ogre.h>
2
3   #include "PremiereApplication.h"
4
5   #if OGRE_PLATFORM == PLATFORM_WIN32 || OGRE_PLATFORM == ↩
        OGRE_PLATFORM_WIN32
6   #define WIN32_LEAN_AND_MEAN
7   #include "windows.h"
8
9   INT WINAPI WinMain(HINSTANCE hInst, HINSTANCE, LPSTR strCmdLine, ↩
        INT)
10  #else
11  int main(int argc, char **argv)
12  #endif
13  {
14
```

```
15      PremiereApplication app;
16
17      try {
18        app.go();
19      } catch(Ogre::Exception& e) {
20  #if OGRE_PLATFORM == OGRE_PLATFORM_WIN32
21          MessageBoxA(NULL, e.getFullDescription().c_str(), "An ↩
                exception has occurred!", MB_OK | MB_ICONERROR | ↩
                MB_TASKMODAL);
22  #else
23          fprintf(stderr, "An exception has occurred: %s\n",
24              e.getFullDescription().c_str());
25  #endif
26      }
27
28      return 0;
29  }
```

3.8 Le viewport

Une zone de rendu est une portion de l'écran sur lequel est affiché ce que voit la caméra. La gestion de l'affichage dans une zone de rendu est à la charge de la classe Viewport.

Cette gestion de la zone de rendu a celà d'important que :

— la façon dont la caméra rend à l'écran ce qu'elle voit ne dépend pas que d'elle. En effet, la taille de votre zone de rendu et son format seront répercutés sur la portion de scène qu'il vous sera donné de voir. Si l'on ne tenait pas compte de ces paramètres, on pourrait obtenir une image aplatie si l'on élargissait la zone de rendu, ou bien au contraire compressée si l'on diminuait la largeur en laissant la hauteur constante.

— sur une même écran, vous pouvez afficher le rendu de plusieurs caméras dans la scène, voire des caméras de différents sceneManager.

Pour cela, nous allons redéfinir la méthode createViewports() qui s'occupait jusqu'alors de ce travail pour nous, et copier la ligne suivante.

```
1       Viewport *vue = mWindow->addViewport(mCamera);
```

Ici, mWindow est la fenêtre de notre application Ogre, c'est une instance de la classe RenderWindow dont nous verrons les détails dans un prochain chapitre.

Grâce à ce Viewport nouvellement créé, nous allons faire coïncider le rapport largeur/hauteur de notre caméra avec celui du Viewport, pour avoir une image non déformée :

```
1  mCamera->setAspectRatio(Real(vue->getActualWidth()) / Real(vue->↩
       getActualHeight()));
```

On applique un cast vers le format Ogre : :Real pour obtenir un ratio décimal. Dans le cas contraire, le ratio serait tronqué pour être entier et la tête de notre ogre favori serait déformée.

Sachez aussi que c'est le Viewport qui définit la couleur de fond de la scène que vous voyez.

```
1  vue->setBackgroundColour(ColourValue(0.0, 0.0, 1.0));
```

Voici donc ma méthode createViewports() au complet :

```
1  void PremiereApplication::createViewports()
2  {
3      Viewport *vue = mWindow->addViewport(mCamera);
4      mCamera->setAspectRatio(Real(vue->getActualWidth()) / Real(↩
           vue->getActualHeight()));
5      vue->setBackgroundColour(ColourValue(0.0, 0.0, 1.0));
6  }
```

Maintenant, notre scène possède un magnifique ciel fond bleu. Image utilisateur

Quatrième partie

La lumière

3.9 Les lumières

3.9.1 Quelques fonctions de base

La lumière est nécessaire pour pouvoir voir quelque chose dans une scène. Comment a-t-on donc pu voir nos objets depuis le début de ce cours ?

Il existe une propriété du scène Manager qui permet de définir une lumière ambiante. Cela permet d'éclairer la scène de façon homogène avec une certaine luminosité. Par défaut, on a un éclairage à la lumière blanche qui permet de voir ce qui se passe dans la scène. La ligne suivante peut être ajoutée au début de la méthode createscène() pour appliquer une lumiŕe ambiante noire nous pourrons ainsi définir des lumiŕes et voir leur influence.

```
1 msceneMgr->setAmbientLight(ColourValue(0.0, 0.0, 0.0));
```

La classe ColourValue permet de définir une couleur en entrant les quantités respectives de rouge, de vert puis de bleu dans un nombre compris entre 0 et 1. Il est aussi possible de définir une composante alpha (la transparence), utile pour des textures par exemple.

Une lumière est un objet de scène, on passe donc par le scène manager pour créer une lumière :

```
1 Light *light = msceneMgr->createLight(''lumiere1'');
```

Par défaut, la lumière créée est de type ponctuelle. Vous avez plus de détails sur les différents types de lumière un peu plus bas.

Mettons tout de suite en place quelques paramètres de base : la couleur émise (diffuse et spéculaire, que nous détaillerons dans le chapitre sur les matériaux) et la position.

```
1 light->setDiffuseColour(1.0, 0.7, 1.0);
2 light->setSpecularColour(1.0, 0.7, 1.0);
3 light->setPosition(-100, 200, 100);
```

Les deux premiers paramètres sont les couleurs diffuses et spéculaires, au format RVB, avec des valeurs qui doivent être comprises entre 0 et 1.

La couleur diffuse est la couleur sous laquelle vont apparaître les objets non brillants, et la couleur spéculaire est un paramètre supplémentaire pour les matériaux réfléchissants comme le métal ou le verre. Pour une lumière, on met généralement la même couleur pour ces deux paramètres.

Vient ensuite la méthode setPosition(), qui ne devrait pas vous poser de problèmes, et enfin une dernière ligne [14] permettant d'amplifier ou de diminuer l'intensité lumineuse. Par défaut, ce coefficient est de 1, mais pour notre scène j'ai voulu l'augmenter pour qu'on y voie un peu plus clair : n'hésitez pas à jouer un peu avec pour faire des essais.

Enfin, sachez qu'il est possible d'attacher une lumière à un noeud de scène. Dans ce cas, la méthode Light : :setPosition() définit la position relative de la lumière par rapport au noeud.

```
1         node->attachObject(light);
```

Vous pouvez donc facilement placer une lumière à la position d'un mesh censé émettre de la lumière - par exemple les phares d'une voiture - et les déplacer en même temps grâce à une seule commande vers le noeud de scène !

14. quelle dernière ligne ? ? ?

3.9.2 Les types de lumières

Ogre peut gérer différents types de lumières selon l'effet désiré. Ils sont au nombre de 3 :

- la lumière ponctuelle : cette lumière émet dans toutes les directions à partir de sa position ;
- la lumière directionnelle : une lumière dont les rayons vont dans une direction unique et qui n'a pas de position. C'est le genre de lumière qui permet de reproduire l'éclairage du soleil par exemple ;
- le projecteur ou spot : c'est une lumière qui émet un cône lumineux à partir de sa position, à la façon d'une lampe-torche.

3.9.3 Lumière ponctuelle

C'est le type de lumière créé par défaut que l'on a vu plus haut. Pour le modifier manuellement, il faut utiliser le type LT_POINT.

```
1         light->setType(Light::LT_POINT);
```

Avec ce type de lumière, il existe une méthode nous permettant aussi de limiter la portée de notre éclairage. Voici le prototype :

```
1 Light::setAttenuation( Real range, Real constant, Real linear, ↩
    Real quadratic )
```

C'est plus délicat car les paramètres doivent être choisis avec soin.

- Le premier est la distance caractéristique d'atténuation, c'est-à-dire la distance à partir de laquelle la luminosité diminue.
- constant est une constante d'atténuation comprise entre 0 et 1. Plus elle est proche de 0, et plus le passage de la lumière à l'ombre est brutal.
- linear et quadratic sont les paramètres de la courbe d'atténuation, et doivent être assez faibles, sinon la lumière s'atténue trop rapidement.

Ca ne marche pas ! L'ogre est bien éclairé mais le sol reste désespérément noir !

L'atténuation ajoute une caractéristique un peu différente pour la gestion de la lumière. En effet, l'éclairage des surfaces est calculé en fonction des vertices situés dans la zone d'éclairage. Lorsqu'un vertice est dans la zone d'éclairage du spot, la surface autour de lui est éclairée, sinon elle est dans l'ombre. Ogre se charge ensuite de faire les dégradés entre les vertices plus ou moins éclairés.

Mais ici, notre plan n'est constitué que de quatre vertices (les coins), dont aucun n'est éclairé par le spot. Le sol n'est donc pas éclairé.

Pour régler ça, il suffit de modifier notre sol pour qu'il possède plus de vertices. Retrouvez la définition du plan et modifiez les paramètres de découpage pour obtenir 10 segments en largeur et en longueur (les deux paramètres avant le true).

```
1 Plane plan(Vector3::UNIT_Y, 0);
2 MeshManager::getSingleton().createPlane(''sol'', ↩
    ResourceGroupManager::DEFAULT_RESOURCE_GROUP_NAME, plan, 500,↩
     500, 10, 10, true, 1, 1, 1, Vector3::UNIT_Z);
```

Notez que plus il y aura de vertices sur le modèle, plus ce sera précis, mais ce sera un peu plus coûteux en ressources.

3.9.4 Lumière directionnelle

Etant donné que cette lumière est de type "soleil" et qu'elle émet à l'infini, il n'est pas utile de renseigner sa position. En revanche, la méthode setDirection() permet de définir le vecteur directeur des rayons lumineux.

```
1 light->setType(Light::LT_DIRECTIONAL);
2 light->setDirection(10.0, -20.0, -5);
```

3.9.5 Projecteur

Le projecteur permet généralement de simuler un éclairage artificiel en proposant une lumière directionnelle définie dans un cône central et un cône extérieur, avec deux intensités différentes. Le cône central définit une lumière plus forte que le cône extérieur, où la lumière est quelque peu atténuée. Ces deux cônes sont définis par leur angle d'ouverture, ainsi que par un falloff, c'est-à-dire un coefficient indiquant si la transition entre les deux cônes doit être plus ou moins rapide :

```
1 light->setType(Light::LT_SPOTLIGHT);
2 light->setPosition(0, 150, -100);
3 light->setDirection(0, -1, 1);
4 light->setSpotlightRange(Degree(30), Degree(60), 1.0);
```

Notez que l'éclairage du spot obéit aux mêmes règles que pour l'atténuation d'une lumière ponctuelle : il faut que les vertices soient éclairés pour que l'éclairage soit visible.

De même que pour les lumières ponctuelles, vous pouvez ajouter une portée limitée à votre projecteur avec la méthode setAttenuation().

3.10 Les ombres

3.10.1 Activer les ombres

Tout d'abord, on doit paramétrer nos lumières et nos entités pour projeter (ou non) des ombres. Que ce soit pour les lumières ou les entités, on utilise la même méthode pour choisir d'activer ou non la projection :

```
1 light->setCastShadows(true);
2 head->setCastShadows(true);
```

Si vous voulez qu'une entité ne projette aucune ombre, il suffit de mettre le paramètre à false. De même si vous voulez qu'une lumière ne projette aucune ombre pour les entités (pour une lumière d'ambiance ou d'ajustement, par exemple).

N'oubliez pas de désactiver la projection d'ombres pour le sol. D'une part parce que celui-ci n'a pas besoin de projeter d'ombres, d'autre part parce que certaines techniques nécessitent d'avoir ce paramètre désactivé pour avoir une ombre sur le mesh.

Avant de pouvoir afficher les ombres, il faut les activer. Cela se fait dans le scène Manager, par exemple :

```
1 msceneMgr->setShadowTechnique(Ogre::SHADOWTYPE_STENCIL_ADDITIVE);
```

On définit ici la technique de rendu qui sera utilisée pour les ombres dans la scène (voir ci-dessous).

3.10.2 Les différents types d'ombres

Ogre permet de générer différents types d'ombres selon les besoins, qui dépendent généralement des modèles concernés par la projection d'ombres.

Il existe deux techniques pour la génération d'ombres :

— le type "Stencil" (pochoir en anglais)

— le type "Texture"

Les ombres de type Stencil sont très précises dans les contours et permettent une très bonne projection d'ombre lorsque l'on y regarde de près. En revanche, elles sont assez coûteuses en ressources, notamment lorsque les mesh sont animés. Enfin, elles ne prennent pas du tout en compte la transparence des textures, un cube de texture transparente projettera donc une ombre si ce paramètre est activé. Les ombres de type Texture permettent de gérer la transparence des textures et sont moins coûteuses en ressources, mais leur précision est plus faible.

Enfin, chacune de ces deux catégories est composée de deux techniques, l'une dite modulative, l'autre additive. On obtient ainsi quatre techniques possibles :

— SHADOWTYPE_TEXTURE_MODULATIVE

— SHADOWTYPE_TEXTURE_ADDITIVE

— SHADOWTYPE_STENCIL_MODULATIVE

— SHADOWTYPE_STENCIL_ADDITIVE

Notez que dans chacun des cas, la technique additive est la meilleure, notamment pour une approche de type Stencil. La différence pour les techniques de type Texture est minime ; en revanche, la technique Stencil additive permet d'obtenir des ombres plus ou moins sombres en fonction de l'éclairage grâce à des passes successives, tandis que la méthode Stencil modulative ne fait que projeter le modèle au sol une seule fois pour chaque lumière.

C'est donc dans le scène Manager que l'on s'occupe de déterminer la technique de rendu des ombres. Par défaut, celles-ci ne sont pas rendues.

```
1 msceneMgr->setShadowTechnique(Ogre::SHADOWTYPE\_STENCIL\↩
      _MODULATIVE);
```

Il n'est possible d'avoir qu'une seule technique enregistrée à la fois dans un scène Manager. Il n'est pas possible de choisir les types d'ombres à générer pour chaque lumière de la scène. Il faut donc faire un choix pour l'ensemble de vos ombres.

Ci-dessous, l'approche basée sur la texture, peu précise mais peu coûteuse en ressources :

Image utilisateur `http://fr.openclassrooms.com/informatique/cours/decouvrez-ogre-3d/les-ombres-1`

Cinquième partie

La gestion des entrées

3.11 Les frame listeners

3.11.1 Des "écouteurs d'images" ?

Utilité

En pratique, le moteur fonctionne dans une boucle, qui ne fait qu'afficher une image, puis fait des calculs ; et ainsi de suite, sans s'arrêter. Il est donc possible pour le programmeur de donner ses instructions avant qu'une image soit rendue, ou bien après, ou bien même pendant que la carte graphique fait le rendu graphique.

Lorsque l'on aura vu comment créer cette boucle de rendu, nous serons à même de donner les instructions de la manière dont nous le désirons. En attendant, je vais vous présenter une classe qui a l'avantage de permettre de faire tout ce que je viens de vous expliquer de façon très simple : le frame listener.

Les méthodes à connaître

Un frame listener est une classe interface qui possède trois méthodes, dont voici les prototypes :

— virtual bool frameStarted(const FrameEvent& evt) ;
— virtual bool frameRenderingQueued(const FrameEvent& evt) ;
— virtual bool frameEnded(const FrameEvent& evt) ;

Chacune de ces méthodes est appelée à un moment précis de la boucle de rendu :

— avant que la frame ne soit rendue (frameStarted)
— après que la frame ait été rendue (frameEnded)
— après que le processeur graphique ait reçu les instructions pour le rendu (frameRenderingQueued)

En créant un objet dérivé de la classe frame listener dans votre application et en réimplémentant ces méthodes virtuelles, vous avez donc la possibilité de demander à Ogre d'effectuer les calculs dont vous avez besoin à chaque image.

frameStarted() et frameEnded() sont très similaires, étant donné qu'elles ont pour seule différence d'être appelées respectivement au début et à la fin de la boucle de rendu. Mais comme on est dans une boucle, en réalité il ne se passe quasiment rien entre l'appel à frameEnded() et celui à frameStarted(). La différence peut être utile par exemple si vous avez un calcul qui semble plus logique d'effectuer après que l'image soit rendue plutôt qu'avant, mais ce n'est qu'une question de lecture du code selon moi.

En revanche la dernière (frameRenderingQueued) est plus subtile. Comme je l'ai dit, elle est appelée dès que la carte graphique reçoit les instructions nécessaires pour afficher l'image à rendre.

Vous le savez probablement, c'est une opération très coûteuse en ressources et c'est souvent ce qui ralentit les jeux vidéo mettant en jeu de nombreux effets graphiques. Pendant ce temps-là, le processeur central attend que ça se passe. Par conséquent, si vous appelez la méthode frameRenderingQueued() pour faire des calculs, vous évitez d'avoir un processeur peu occupé pendant que la carte graphique fait son boulot !

De manière générale, on pourra utiliser cette méthode pour des opérations lourdes dont on sait qu'elles seront répétées à chaque image, afin de rentabiliser

l'utilisation du processeur.

Contrôle de l'exécution

La valeur de retour des méthodes d'un frame listener est un booléen récupéré par Ogre pour savoir s'il doit continuer ou non l'exécution du programme.

Tant que cette valeur est true, le rendu continue. En revanche, si une des méthodes renvoie la valeur false, l'exécution s'interrompt et le programme se ferme.

Utiliser plusieurs frame listeners

Il est possible de créer autant de frame listeners que vous le désirez, pour effectuer des opérations diverses. En revanche, il est conseillé de ne pas en abuser pour éviter de trop segmenter votre application, il peut être intéressant d'appeler d'autres fonctions à partir d'un frame listener plutôt que d'en créer trop.

Enfin, et c'est là le plus important : **L'ordre d'exécution des frame listeners est laissé aux soins du moteur. Vous n'avez AUCUN contrôle dessus !**

En d'autres termes, si vous avez besoin d'effectuer des opérations dans un ordre précis, ne les mettez pas dans des frame listeners différents, car vous ne pourrez pas décider de l'ordre d'exécution. Il faut alors laisser un seul frame listener gérer les opérations, ou ne pas passer par eux (ce sera possible lorsque nous attaquerons la boucle de rendu).

3.11.2 Le frame listener en pratique

Comme nous allons vouloir redéfinir les méthodes du frame listener, il faut en faire une classe dérivée. Vu que nous sommes dans la gestion des entrées, nous allons tout de suite préparer le terrain en créant une classe InputListener dérivant de ExampleFrameListener.

Je dérive ici de la classe ExampleFrameListener, elle-même dérivée de FrameListener, car la classe ExampleApplication met déjà en place une gestion des entrées et attend un ExampleFrameListener. Cette classe s'occupe aussi de construire les objets nécessaires à l'écoute des entrées souris/clavier, ce que nous aborderons ultérieurement. Cependant, la méthode de traitement reste identique, nous allons juste devoir redéfinir frameRenderingQueued() pour implémenter notre propre gestion des entrées à la place de celle prévue par ExampleFrameListener.

Listing 3.6 – ExampleFrameListener.cpp

```
1 #include ''ExampleFrameListener.h''
2 class InputListener: public ExampleFrameListener
3 {
4 public:
5     InputListener(RenderWindow* win, Camera* cam, sceneManager *↩
          sceneMgr, bool bufferedKeys = false, bool bufferedMouse =↩
           false, bool bufferedJoy = false );
6     virtual bool frameRenderingQueued(const FrameEvent& evt);
7
8 private:
```

```
9       Ogre::sceneManager *msceneMgr; //pointeur sur le scene ↩
            Manager, qui servira a retrouver des objets dans la scene↩
            .
10      bool mToucheAppuyee;          //pour garder une trace de l ↩
            etat dans lequel se trouve une touche particuliere.
11
12      /*distance de deplacement de la camera et sa vitesse, puis ↩
            des angles de rotation.*/
13      Ogre::Real mMouvement;
14      Ogre::Real mVitesse;
15      Ogre::Real mVitesseRotation;
16
17          /*angles de rotation.*/
18      Ogre::Radian mRotationX;
19      Ogre::Radian mRotationY;
20  };
```

Dans votre fichier InputListener.cpp, préparez le constructeur ainsi que l'implémentation des méthodes, avec un corps vide pour l'instant :

Listing 3.7 – InputListener.cpp

```
1   /*
2   Arguments:
3   RenderWindow* win: votre RenderWindow pour l'application
4   Camera* cam: la camera que vous utilisez
5   sceneManager *sceneMgr: le scene Manager
6   bool bufferedKeys: indique si vous desirez utiliser le buffer ↩
        pour le clavier
7   bool bufferedMouse: indique si vous desirez utiliser le buffer ↩
        pour la souris
8   bool bufferedJoy: indique si vous desirez utiliser le buffer pour↩
         le joystick
9   */
10  InputListener::InputListener(RenderWindow* win, Camera* cam, ↩
        sceneManager *sceneMgr, bool bufferedKeys, bool bufferedMouse↩
        , bool bufferedJoy)
11        : ExampleFrameListener(win, cam, bufferedKeys, ↩
              bufferedMouse, bufferedJoy)
12  {
13      msceneMgr = sceneMgr;
14      mVitesse = 100;
15      mVitesseRotation = 0.3;
16      mToucheAppuyee = false;
17  }
```

Il n'y a pas de constructeur écrit pour les frame listeners, car c'est une classe interface, seules les trois méthodes que j'ai présentées plus haut importent. En revanche, la classe ExampleFrameListener possède un constructeur qui prépare le terrain pour l'utilisation des entrées, que j'appelle dans ma classe dérivée.

Les trois booléens en paramètres indiquent si vous désirez utiliser le buffer respectivement pour le clavier, la souris et le joystick. Comme ce sera l'objet de la dernière partie de ce chapitre, je mets par défaut false (dans le .h).

Dans le corps du constructeur, j'initialise mes attributs mSceneMgr, mVitesse et mToucheAppuyee, qui nous serviront par la suite.

Il n'y a pas d'attribut à ajouter dans notre classe PremiereApplication, la classe ExampleApplication contient déjà un pointeur sur un ExampleFrameListener.

3.12 OIS

Pour gérer les entrées de l'utilisateur, nous allons utiliser la bibliothèque OIS [15], qui est distribuée par défaut avec le SDK d'Ogre. Comme je l'ai dit en introduction de ce cours, un moteur 3D n'a pas de méthodes pour gérer autre chose que ce qui s'affiche sur votre écran. C'est pourquoi nous utiliserons cette bibliothèque pour récupérer les actions du joueur.

Pour ce faire, OIS représente les périphériques d'entrée par des objets, qui sont les suivants :
— Mouse pour la souris ;
— Keyboard pour le clavier ;
— Joystick pour les joysticks ou manettes de jeu.

Qui dit nouvelle bibliothèque dit nouveau namespace ! Ces classes se trouvent donc dans l'espace de nom OIS.

Les touches du clavier et les boutons de la souris sont des énumérations, définies comme ceci :
— OIS : :KC_NOMDELATOUCHE pour le clavier ;
— OIS : :MB_NOMDUBOUTON pour la souris.
Ce qui donne par exemple OIS : :KC_A pour la touche 'A' [16], ou OIS : :MC_Left pour le clic gauche.

Une dernière chose bonne à savoir : les codes de touches d'OIS correspondent aux touches physiques d'un clavier QWERTY. Ce qui signifie par exemple que OIS : :KC_A correspond à la touche Q sur votre clavier AZERTY.

Afin d'utiliser OIS, il faut inclure le header correspondant. Celui-ci se trouve dans le dossier OIS du dossier include, on ajoutera donc la ligne de préprocesseur suivante en tête du header de InputListener :

Listing 3.8 – Include OIS

```
1 #include <OIS/OIS.h>
```

Bien sûr, vous pouvez aussi ajouter le répertoire OIS à la liste des includes de votre IDE pour éviter d'avoir à le préciser dans le code.

3.13 Allons-y sans buffer

3.13.1 Explications

Dans le constructeur de notre frame listener, je vous ai dit que l'on avait mis les paramètres concernant l'utilisation du buffer à false, car c'est l'objet de la dernière partie de ce chapitre. Mais que signifie le fait d'utiliser ou non le buffer ?

15. Object Oriented Input System, c'est à dire : Système d'entrées orienté objet
16. la touche 'Entrée' s'appelle 'Return' en anglais, ne cherchez donc pas OIS : :KC_ENTER, vous ne trouverez pas.

Lorsque vous appuyez sur une touche, le clavier envoie un signal à l'ordinateur pour lui dire qu'une touche est actuellement pressée, en précisant quelle touche est concernée. Ce signal est envoyé tant que la touche reste enfoncée.

De son coté, notre programme effectue sa boucle infinie, rendant les images et faisant les calculs demandés. Supposons que vous appuyez sur une touche à un instant donné. Lorsque l'ordinateur arrivera à l'instruction lui demandant de regarder ce qui se passe sur le clavier, il va voir qu'une touche est enfoncée et cherchera à effectuer les opérations demandées, et ceci tant que la touche reste enfoncée.Mais il n'est pas possible pour l'ordinateur de faire seul la différence entre une touche enfoncée et une touche qui vient d'être enfoncée.

Nous allons donc voir comment régler ce problème "à la main", puis nous verrons l'utilisation du buffer, qui constitue une autre façon de traiter l'entrée.

3.13.2 Création du frame listener

Avant de passer à la suite, réimplémentez la méthode createFrameListener() présente dans ExampleApplication dans la classe PremiereApplication en ajoutant le prototype et la définition :

Listing 3.9 – InputListener.cpp

```
1 void PremiereApplication::createFrameListener()
2 {
3     //creation du framelistener en utilisant le ctor prepare plus↩
          tot
4     mFrameListener= new InputListener(mWindow, mCamera, mSceneMgr↩
          , false, false, false);
5     //signale a l objet root que ns avons un nv frame listener et↩
          qu il faudra l'appeler
6     mRoot->addFrameListener(mFrameListener);
7 }
```

Le root est l'élément de base de l'application Ogre qui s'occupe notamment de gérer les frame listeners, nous le reverrons plus tard en approfondissant le fonctionnement du moteur.

3.13.3 Déplacer la caméra

Tout d'abord, nous allons considérer le déplacement de caméra, pour lequel nous n'avons pas besoin de savoir si la touche vient d'être appuyée : c'est simplement son état actuel qui compte.

Premièrement, il nous faut récupérer l'état actuel du clavier et de la souris. Localisez la méthode frameRenderingQueued() de votre InputListener et insérez-y ceci :

```
1 if(mMouse)
2     mMouse->capture();
3 if(mKeyboard)
4     mKeyboard->capture();
```

Ces deux lignes permettent de mettre à jour nos objets pour obtenir le nom des touches enfoncées.

Vérifions d'abord si la touche Echap est utilisée, auquel cas nous quitterons l'application.

```
1  if(mKeyboard->isKeyDown(OIS::KC_ESCAPE))
2      return false;
```

Nous devons ensuite mettre à jour la valeur de mMouvement, qui sera la distance parcourue par la caméra si une direction est choisie. Comme le nombre d'images par seconde est variable, nous utilisons la propriété timeSinceLastFrame de l'événement, multipliée par la vitesse de la caméra. Le produit de la vitesse par le temps écoulé nous donne donc la distance parcourue.

J'ai aussi créé un vecteur dans lequel nous allons enregistrer les déplacements à effectuer. En effet, on peut utiliser plusieurs touches en même temps, il faut donc additionner les directions demandées, et les conserver pour déplacer la caméra en une seule fois.

```
1  Ogre::Vector3 deplacement = Ogre::Vector3::ZERO;
2  mMouvement = mVitesse * evt.timeSinceLastFrame;
```

Nous allons utiliser les flèches du clavier et les touches Z, S, Q, D pour nous déplacer ; j'ai aussi implémenté les touches fléchées, qui sont une configuration alternative pour le déplacement. Il faut donc vérifier si les touches qui nous intéressent sont enfoncées :

```
1  // La touche A d'un clavier QWERTY correspond au Q sur un AZERTY
2  if(mKeyboard->isKeyDown(OIS::KC_LEFT) || mKeyboard->isKeyDown(OIS↩
       ::KC_A))
3      deplacement.x -= mMouvement;
4
5  if(mKeyboard->isKeyDown(OIS::KC_RIGHT) || mKeyboard->isKeyDown(↩
       OIS::KC_D))
6      deplacement.x += mMouvement;
7
8  // W correspond au Z du AZERTY
9  if(mKeyboard->isKeyDown(OIS::KC_UP) || mKeyboard->isKeyDown(OIS::↩
       KC_W))
10     deplacement.z -= mMouvement;
11
12 if(mKeyboard->isKeyDown(OIS::KC_DOWN) || mKeyboard->isKeyDown(OIS↩
       ::KC_S))
13     deplacement.z += mMouvement;
```

Attention aux signes ! Vous devez respecter ce que nous avons vu dans le chapitre sur les déplacements ![17]

Le déplacement de la caméra fonctionne, il ne manque plus que la rotation de celle-ci. Il nous suffit pour cela de récupérer le déplacement relatif depuis la dernière fois que la souris a bougé (depuis le dernier appel à frameRenderingQueued() donc).

Pour retrouver cette valeur, on passe successivement par les attributs suivants :

- le mouseState contenu dans l'objet souris, contenant diverses informations sur l'état de la souris ;
- l'axe que l'on désire observer : ici, ce sera X ou Y pour le yaw ou le pitch ;
- le déplacement relatif de la souris suivant cet axe.

Maintenant, occupons-nous du mouvement de la souris. Pour récupérer le déplacement de celle-ci, nous devons récupérer son état, comme indiqué dans le code suivant.

17. hein ? de quoi parle t il ?

```
const OIS::MouseState &mouseState = mMouse->getMouseState();
```

À partir de cette référence on peut notamment récupérer le déplacement de la souris depuis la dernière image, en appelant l'axe X ou Y puis l'attribut rel.

```
mRotationX = Degree(-mouseState.Y.rel * mVitesseRotation);
mRotationY = Degree(-mouseState.X.rel * mVitesseRotation);
```

Il faut particulièrement faire attention aux axes et aux signes. Je considère que mRotationX (respectivement mRotationY) correspond à la rotation autour de l'axe X (respectivement Y), c'est-à-dire lorsque je déplace ma souris en avant ou en arrière (respectivement à gauche ou à droite). Or, le déplacement vers l'avant ou l'arrière de la souris correspond à son axe Y, c'est pour ça que je demande l'axe Y de la souris pour trouver la rotation autour de X dans l'espace 3D.

On rajoute une multiplication par la vitesse de rotation voulue et on convertit le tout en degrés, sinon le mouvement est bien trop rapide.

Enfin, on appelle les méthodes de rotation et de déplacement de la caméra :

```
mCamera->yaw(mRotationY);
mCamera->pitch(mRotationX);
mCamera->moveRelative(deplacement);
```

La dernière ligne, comme vous le remarquez, déplace la caméra par rapport à son repère local, ce qui évite de faire la transformation de la variable deplacement à la main. Il est aussi possible de demander un déplacement par rapport au repère absolu avec Camera : :move().

3.13.4 Et avec un noeud de scène ?

J'en profite pour vous montrer comment on aurait procédé pour déplacer un noeud de scène par exemple, qui utilise la méthode translate().

Les deux lignes suivantes sont équivalentes :

```
node->translate(deplacement, TS_LOCAL);
node->translate(node->getOrientation() * deplacement, TS_PARENT);
```

La première ligne est très similaire à celle utilisée pour la caméra, il suffit de préciser que l'on se déplace par rapport au repère local du noeud de scène.

La seconde solution indique un déplacement relatif au noeud parent, mais utilise le quaternion retourné par la méthode getOrientation() multiplié par le vecteur de déplacement pour obtenir la direction souhaitée dans ce repère. En pratique, on utilisera seulement la première ligne, plus courte et plus propre dans le code.

3.13.5

Mini-TP : créer un interrupteur

Pour gérer un événement qui ne doit arriver qu'une fois lorsque la touche est appuyée, il y a une précaution supplémentaire à prendre. Je vous ai dit plus haut que votre ordinateur ne retenait pas l'état dans lequel se trouvait votre clavier ou votre souris à l'image précédente. Cependant, rien ne nous empêche de le faire nous-mêmes !

À titre d'exemple, disons que l'on veut utiliser la touche T pour allumer et éteindre la lumière de notre scène. Il va donc falloir vérifier à chaque image si la touche T est enfoncée, et si en plus ce n'était pas déjà le cas à l'image précédente. Pour cela, on utilisera l'attribut mToucheAppuyee de notre classe InputListener.

Un indice : le SceneManager possède une méthode getLight() qui permet de récupérer un pointeur sur une lumière à partir du nom de celle-ci...

À vos claviers ! La réponse se trouve juste après.

Listing 3.10 – Capture de l'état ponctuel d'une touche

```
1 bool etatTouche = mKeyboard->isKeyDown(OIS::KC_T);
2 if(etatTouche && !mToucheAppuyee)
3 {
4     Ogre::Light *light = mSceneMgr->getLight("lumiere1");
5     light->setVisible(!light->isVisible());
6 }
7 mToucheAppuyee = etatTouche;
```

Tout d'abord, je récupère l'état actuel de ma touche T dans une variable locale. Je vérifie si la touche est actuellement enfoncée et si elle ne l'était pas déjà à l'aide de l'attribut mToucheAppuyee. Si ma condition est vérifiée, je récupère ma lumière, et je change son état (visible ou non).

Enfin, j'enregistre l'état actuel de ma touche T dans mToucheAppuyee, en prévision de la prochaine image !

3.14 Avec buffer, c'est plus simple ?

3.15 Avec buffer, c'est plus simple ?

La méthode présentée ci-avant pour contrÃ´ler si une touche vient ou non d'être appuyée fonctionne mais est peut pratique si on doit surveiller quinze touches.

Heureusement, OIS a pensé à tout, nous allons donc voir une autre façon de faire ce que l'on vient juste d'écrire. On va commencer comme précédemment par le déplacement de la caméra, puis on verra comment gérer notre interrupteur.

3.15.1 Mise en place

Nous allons commencer par activer l'utilisation du buffer pour la souris et le clavier lors de la construction de notre frame listener. Il suffit pour cela de mettre les paramètres correspondants à true.

Listing 3.11 – Activation du buffer pour la souris et le clavier

```
1 void PremiereApplication::createFrameListener()
2 {
3     mFrameListener= new InputListener(mWindow, mCamera, mSceneMgr↩
          , true, true, false);     mRoot->addFrameListener(↩
          mFrameListener);
4 }
```

C'est quasiment tout , il va simplement falloir rajouter deux petites lignes dans le constructeur pour que tout soit prêt.

Afin d'utiliser le buffer, il faut fournir un objet (un Â≪ écouteur Â≫ dérivant d'une des classes OIS : :***Listener selon le périphérique à écouter) qui sera celui qui recevra les événements du type "cette touche vient d'être appuyée, que dois-je faire ?". Pour cela, OIS fournit une méthode pour chacun de trois périphériques d'entrée (clavier, souris, joystick) :

— virtual void OIS : :Mouse : :setEventCallback(OIS : :MouseListener* mouseListener) ;
— virtual void OIS : :Keyboard : :setEventCallback(OIS : :KeyListener* keyListener) ;
— virtual void OIS : :JoyStick : :setEventCallback(OIS : :JoyStickListener* joystickListener) ;

Cette méthode prend donc en paramètre un pointeur sur un listener du périphérique que vous voulez utiliser. Nous allons donc rajouter deux classes mères à notre InputListener : OIS : :MouseListener et OIS : :KeyListener. Modifiez donc la déclaration de la classe InputListener :

Listing 3.12 – Classes mères pour gestion des Listeners

```
1 class InputListener : public ExampleFrameListener, OIS::↩
      KeyListener, OIS::MouseListener
```

Dans le constructeur, vous pouvez maintenant insérer les deux lignes suivantes (mMouse et mKeyboard sont déclarées dans ExampleFrameListener) :

Listing 3.13 – Enregistrement des listener

```
1 mMouse->setEventCallback(this);
2 mKeyboard->setEventCallback(this);
```

Vous ne pouvez enregistrer qu'un seul écouteur par périphérique d'entrée. En cas d'appels multiples à la méthode setEventCallback(), c'est le dernier appel qui définit l'écouteur à utiliser. Pour que différents objets reçoivent les événements, il faudra donc les redistribuer à partir de l'écouteur receveur.

Au chapitre des modifications, supprimez les anciens attributs d'InputListener et mettez ceux-ci :

Listing 3.14 – Attributs d'InputListener

```
1 private:
2     Ogre::SceneManager *mSceneMgr;
3     bool mContinuer;
4     Ogre::Vector3 mMouvement;
5     Ogre::Real mVitesse;
6     Ogre::Real mVitesseRotation;
```

En initialisant ces attributs, votre constructeur devrait maintenant ressembler à ceci :

Listing 3.15 – Constructeur d'InputListener

```
1 InputListener(RenderWindow* win, Camera* cam, SceneManager *↩
      sceneMgr, bool bufferedKeys = false, bool bufferedMouse = ↩
      false, bool bufferedJoy = false )    : ExampleFrameListener(↩
      win, cam, bufferedKeys, bufferedMouse, bufferedJoy)
2 {
3     mSceneMgr = sceneMgr;
4     mContinuer = true;  //permettra d'enregistrer l'appui sur la ↩
          touche Echap
```

```
5       mMouvement = Ogre::Vector3::ZERO; //vecteur de la direction ds↩
            lqulle se d\'eplacer
6       mVitesse = 100;
7       mVitesseRotation = 0.2; //facteur multiplicatif pr ajuster la ↩
            vitesse de la cam
8       mMouse->setEventCallback(this);
9       mKeyboard->setEventCallback(this);
10  }
```

L'attribut mContinuer permettra d'enregistrer l'appui sur la touche Echap, mMouvement sera le vecteur de la direction dans laquelle on doit se déplacer et mVitesseRotation un facteur multiplicatif permettant d'ajuster la vitesse de rotation de la caméra.

On met donc à jour la valeur de retour de la méthode frameRenderingQueued().

```
1   bool InputListener::frameRenderingQueued(const Ogre::FrameEvent& ↩
        evt)
2   {
3       if(mMouse)
4           mMouse->capture();
5       if(mKeyboard)
6           mKeyboard->capture();
7
8       return mContinuer;
9   }
```

Enfin, il y a des méthodes virtuelles pures à réimplémenter dans notre classe. Ces méthodes seront appelées lors de l'événement correspondant sur le clavier (touche enfoncée ou relâchée) ou sur la souris (bouton appuyé ou relâché, déplacement). De même que les méthodes des frame listeners d'Ogre, elles renvoient un booléen que l'on utilisera pour savoir si l'on doit interrompre le programme.

Ajoutons donc les prototypes dans notre header et un simple retour de valeur dans le corps des méthodes pour commencer.

Contrairement aux méthodes des frame listeners, la valeur de retour ne détermine pas si l'on doit continuer ou non l'exécution. C'est pour cela que l'on devra passer par l'attribut mContinuer pour surveiller l'appui sur la touche Echap.

Listing 3.16 – Méthodes virtuelles appelées lors d'un évènement sur un périphérique

```
1   bool InputListener::mouseMoved(const OIS::MouseEvent &e)
2   {
3       return true;
4   }
5
6   bool InputListener::mousePressed(const OIS::MouseEvent &e, OIS::↩
        MouseButtonID id)
7   {
8       return true;
9   }
10
11  bool InputListener::mouseReleased(const OIS::MouseEvent &e, OIS::↩
        MouseButtonID id)
12  {
13      return true;
```

```
14  }
15
16  bool InputListener::keyPressed(const OIS::KeyEvent &e)
17  {
18      return true;
19  }
20
21  bool InputListener::keyReleased(const OIS::KeyEvent &e)
22  {
23      return true;
24  }
```

On commence par implémenter la touche Echap. Si elle est appuyée, on passe simplement l'attribut mContinuer à false.

Listing 3.17 – Implémentation de l'appuie sur ECHAP

```
1   bool InputListener::keyPressed(const OIS::KeyEvent &e)
2   {
3       switch(e.key)
4       {
5           case OIS::KC_ESCAPE:
6               mContinuer = false;
7               break;
8       }
9
10      return mContinuer;
11  }
```

On gère ensuite l'appui sur les touches de déplacement en modifiant les composantes de mMouvement en fonction de la touche. On va aussi multiplier la vitesse de déplacement par deux lorsque l'on appuie sur la touche majuscule gauche.

Listing 3.18 – Implémentation de l'appuie sur les touches de déplacement

```
1   bool InputListener::keyPressed(const OIS::KeyEvent &e)
2   {
3       switch(e.key)
4       {
5           case OIS::KC_ESCAPE:
6               mContinuer = false;
7               break;
8           case OIS::KC_W:
9               mMouvement.z -= 1;
10              break;
11          case OIS::KC_S:
12              mMouvement.z += 1;
13              break;
14          case OIS::KC_A:
15              mMouvement.x -= 1;
16              break;
17          case OIS::KC_D:
18              mMouvement.x += 1;
19              break;
20          case OIS::KC_LSHIFT:
21              mVitesse *= 2;
22              break;
23      }
24
25      return mContinuer;
26  }
```

Enfin, dans la méthode keyReleased, on va "retirer" la composante que l'on ajoute lors de l'appui sur une touche. Le code est donc semblable, seuls les signes changent.

```
bool InputListener::keyReleased(const OIS::KeyEvent &e)
{
    switch(e.key)
    {
        case OIS::KC_W:
            mMouvement.z += 1;
            break;
        case OIS::KC_S:
            mMouvement.z -= 1;
            break;
        case OIS::KC_A:
            mMouvement.x += 1;
            break;
        case OIS::KC_D:
            mMouvement.x -= 1;
            break;
        case OIS::KC_LSHIFT:
            mVitesse /= 2;
            break;
    }

    return true;
}
```

Maintenant que l'on gère correctement l'évolution de nos variables de mouvement et de vitesse de déplacement, il faut écrire le déplacement de la caméra dans la méthode frameRenderingQueued().

Listing 3.19 – Implémentation du déplacement de la caméra dans la méthode frameRenderingQueued

```
virtual bool frameRenderingQueued(const FrameEvent& evt)
{
    if(mMouse)
        mMouse->capture();

    if(mKeyboard)
        mKeyboard->capture();

    Ogre::Vector3 deplacement = Ogre::Vector3::ZERO;
    deplacement = mMouvement * mVitesse * evt.timeSinceLastFrame;
    mCamera->moveRelative(deplacement);

    return mContinuer;
}
```

Pour la rotation de la caméra, tout se passe dans la méthode mouseMoved(), dont l'événement reçu en paramètre contient l'état de la souris, permettant comme précédemment de retrouver le déplacement relatif de la souris.

On multiplie cette valeur relative par la vitesse de rotation, on fait attention aux signes, et voici ce qu'on obtient :

Listing 3.20 – Implémentation de la rotation de la caméra dans la méthode mouseMoved

```
bool InputListener::mouseMoved(const OIS::MouseEvent &e)
{
```

```
3       mCamera->yaw(Ogre::Degree(-mVitesseRotation * e.state.X.rel))↩
            ;
4       mCamera->pitch(Ogre::Degree(-mVitesseRotation * e.state.Y.rel↩
            ));
5
6       return true;
7   }
```

Vous pouvez maintenant compiler et exécuter votre application ; les commandes de déplacement sont maintenant gérées entièrement par notre classe InputListener, n'hésitez donc pas à adapter les variables initialisées dans le constructeur si vous voulez accélérer ou ralentir les mouvements par exemple.

Cette méthode de gestion des entrées permet donc de gérer plus facilement l'appui ponctuel sur une touche, tout en conservant une gestion simple des touches qui peuvent rester enfoncées (pour le déplacement ici).

Gardez cependant bien à l'esprit qu'**il ne peut y avoir qu'un seul écouteur par périphérique d'entrée** et qu'il faudra donc penser à rapporter l'appui sur les touches à des écouteurs annexes lorsque votre application grossira, sinon vous allez vite vous retrouver avec un code lourd et mal organisé.

Dans ce chapitre, nous avons vu comment gérer nous-mêmes les entrées de l'utilisateur avec la bibliothèque OIS, ainsi que le principe des frame listeners, qui nous ont ici été bien utiles alors que nous n'avons pas encore vu le fonctionnement de la boucle de rendu.

Le prochain chapitre promet d'être intéressant : nous allons en effet utiliser le module de terrain d'Ogre qui a été refait dans la version 1.7 et qui offre une gestion beaucoup plus optimisée des terrains par rapport aux versions précédentes. Je n'en dis pas plus, on se retrouve de l'autre côté.

Sixième partie

Gardez les pieds sur Terre

Sans outil particulier, créer un terrain pourrait relever d'un travail de longue haleine si l'on devait créer le mesh dans les moindres détails pour l'insérer ensuite à la scène. La gestion d'une telle entité serait aussi relativement complexe, puisqu'il doit pouvoir interagir avec tous les objets susceptibles de se déplacer dans la scène.

Heureusement, plutôt que de placer notre terrain comme une simple entité dans la scène, Ogre propose de passer par une classe Terrain qui, comme son nom l'indique si bien, permet de gérer des terrains dans la scène.

Le terrain n'est généralement pas seul et le ciel joue un rôle important pour le réalisme de la scène. Là encore, quelques outils bienvenus offrent différentes solutions pour obtenir un résultat convaincant.

3.16 Créer un terrain

3.16.1 Préparation

Avant de commencer, il va falloir modifier un peu notre projet avec de nouvelles dépendances pour que les terrains soient utilisables.

Dans l'éditeur de lien, ajoutez le fichier librairie OgreTerrain.lib (ou bien OgreTerrain_d.lib si vous compilez en debug).

Il faut maintenant ajouter un fichier en-tête dans notre classe :

Listing 3.21 – Ajout du fichier d'entête pour la gestion des terrains

```
1 #include <Ogre/Terrain/OgreTerrain.h>
```

Enfin, si vous ne les avez pas déjà placées dans le répertoire de votre exécutable, copiez ces 2 DLL dans le dossier correspondant à votre configuration (les DLL de debug se terminent aussi par "_d") :

- OgreTerrain.dll
- OgrePaging.dll

3.16.2 Quelques paramètres à régler

Ajout d'attributs obligatoires pour le terrain

Commençons par ajouter deux attributs dans notre classe PremiereApplication.

- un objet Terrain, qui gérera les propriétés de notre terrain,
- un objet TerrainGlobalOptions, qui définit des propriétés générales pour les terrains dans notre application, notamment l'éclairage.

La présence de cet objet TerrainGlobalOptions est obligatoire lorsque vous voulez utiliser les terrains dans votre scène, sous peine d'erreur à l'exécution.

Listing 3.22 – Attributs pour la gestion de terrain

```
1 Ogre::Terrain *mTerrain;
2 Ogre::TerrainGlobalOptions *mGlobals;
```

Au début de la méthode createScene(), nous allons régler quelques paramètres pour que notre terrain apparaisse sous son meilleur jour.

Réglage de la caméra

Premièrement, je vous conseille d'augmenter la distance de vue de la caméra et de la positionner en hauteur :

Listing 3.23 – Réglage de la caméra

```
1 mCamera->setFarClipDistance(20000);
2 mCamera->setPosition(0, 500, 0);
```

Il est aussi possible de régler la distance de vue à l'infini, en mettant 0 comme paramètre. Cependant, cela dépend de votre machine, il faut donc vérifier si vous pouvez vous le permettre avant de l'appliquer.

Listing 3.24 – Vérification et réglages de vue à l'infini

```
1 if (mRoot->getRenderSystem()->getCapabilities()->hasCapability(↩
      Ogre::RSC\_INFINITE\_FAR\_PLANE))
2     mCamera->setFarClipDistance(0);
```

Définition des éclairages

Plutôt que de gérer l'éclairage de façon distincte, on peut définir un éclairage d'ambiance particulier pour le terrain. Pour cela, il suffit de définir une lumière directionnelle avec les paramètres qui vous paraissent adaptés à votre environnement.

Ici cette lumière est ajoutée en tant qu'attribut de la classe, car je l'utiliserai dans une autre fonction.

Listing 3.25 – Définition de l'éclairage pour le terrain

```
1 Ogre::Vector3 lightdir(0.55f, -0.3f, 0.75f);
2 mLight = mSceneMgr->createLight(''terrainLight'');
3 mLight->setType(Ogre::Light::LT\_DIRECTIONAL);
4 mLight->setDirection(lightdir);
5 mLight->setDiffuseColour(Ogre::ColourValue::White);
6 mLight->setSpecularColour(Ogre::ColourValue(0.4f, 0.4f, 0.4f));
```

Définition d'une méthode pour la prise en charge de l'initialisation du terrain

Nous allons définir une méthode createTerrain(), qui sera appelée dans createScene() et qui prendra en charge toute l'initialisation du terrain.

À l'intérieur de celle-ci, commencez par créer le TerrainGlobalOptions :

Listing 3.26 – Méthode pour la prise en charge de l'initialisation du terrain

```
1 void PremiereApplication::createTerrain()
2 {
3     mGlobals = OGRE_NEW Ogre::TerrainGlobalOptions();
4     mGlobals->setMaxPixelError(8);
5 }
```

mGlobals est le premier objet d'Ogre que nous allons créer nous-mêmes, sans passer par l'intermédiaire du Scene Manager.

Le moteur fournit différentes macros comme OGRE_NEW pour allouer l'espace en mémoire lorsque vous instanciez une classe d'Ogre. De façon générale,

les développeurs conseillent d'utiliser ces macros lorsque vous devez faire de l'allocation dynamique sur les objets du moteur qui dérivent de Ogre : :AllocatedObject. Vous pouvez voir la hiérarchie des classes dans la documentation.

En ce qui concerne la destruction des objets, l'utilisation de l'opérateur OGRE_NEW implique l'utilisation de l'opérateur OGRE_DELETE pour libérer l'espace mémoire.

La seconde ligne appelle la méthode setMaxPixelError() qui donne la précision avec laquelle le terrain est rendu. La valeur indiquée est l'erreur tolérée, en pixels, pour l'affichage du terrain. Plus la valeur est faible, plus le terrain correspond au modèle donné, plus elle est forte et plus il sera imprécis, en donnant l'impression d'un terrain nivelé.

On applique maintenant les réglages concernant l'éclairage à nos options globales.

Listing 3.27 – Application des réglages

```
1 mGlobals->setLightMapDirection(mLight->getDerivedDirection());
2 mGlobals->setCompositeMapDistance(3000);
3 mGlobals->setCompositeMapAmbient(mSceneMgr->getAmbientLight());
4 mGlobals->setCompositeMapDiffuse(mLight->getDiffuseColour());
```

3.16.3 Le terrain

Maintenant, passons à la création du terrain lui-même. Comme pour les options du terrain, nous ne passons pas par une méthode du Scene Manager pour créer le terrain, mais par le constructeur de la classe, qui prend tout de même en paramètre le Scene Manager de votre scène.

Listing 3.28 – Création du terrain

```
1 mTerrain = OGRE_NEW Ogre::Terrain(mSceneMgr);
```

Il faut à présent définir précisément les paramètres de notre terrain :

— son relief,

— sa taille,

— ses textures.

Il est temps que je vous parle des heightmaps pour la modélisation du terrain !

Les heightmaps

Une heightmaps est une image qui contient des informations de relief. On les utilise pour stocker le relief d'un terrain, mais aussi pour connaitre le relief d'une texture, afin de donner l'impression de relief sur un mesh alors qu'en réalité il est plat ou simplement lisse (c'est le principe du bump mapping).

Cette méthode a le principal avantage d'être très légère en terme de stockage, puisque l'on a simplement une image à la place d'un modèle 3D complet qui prendrait beaucoup plus d'espace à stocker.

Voici l'image que nous allons utiliser pour notre terrain : Image utilisateur

Comme vous le voyez c'est une simple image en noir et blanc, et pourtant cela suffit amplement !

En effet, si l'on utilise uniquement des niveaux de gris dans une image, chaque pixel peut prendre 256 valeurs, 0 correspondant au noir et à la hauteur la plus faible, 255 au blanc et à la plus forte altitude. On a donc 256 altitudes

possibles pour notre terrain, ce qui est tout à fait honnête et suffit à la majorité des cas.

En réalité, chaque pixel possède trois valeurs, correspondant à la quantité de rouge, de vert et de bleu, chacune de ces valeurs allant de 0 à 255. Or pour les niveaux de gris, ces trois valeurs doivent être identiques, ce qui laisse 255 triplets de valeurs : (0, 0, 0) pour le noir, puis les niveaux de gris et enfin (255, 255, 255) pour le blanc).

Lors de la création d'un fichier heightmap, on fait en sorte que le point le plus haut de notre carte soit blanc et que le point le plus bas soit noir, afin d'utiliser toute la plage de valeurs disponibles dans la carte et éviter les dénivellations peu naturelles.

Pour charger un fichier heightmap, on passe par un objet Image qui va chercher le nom du fichier que vous voulez dans les ressources déjà chargées. J'utilise le fichier terrain.png, que vous pouvez trouver dans "OgreSDK.media.materials.textures". [18]

Listing 3.29 – Chargement du fichier heightmap

```
1 Ogre::Image img;
2 img.load(''terrain.png'', Ogre::ResourceGroupManager::↩
    DEFAULT_RESOURCE_GROUP_NAME);
```

Les paramètres géométriques

Pour fournir toutes les informations dont le terrain a besoin pour être généré, on utilise sa méthode prepare() qui prend en paramètre un Terrain : :Import-Data, qui est en gros une classe contenant l'ensemble des paramètres à fournir au terrain. On va donc commencer par créer cet objet :

Listing 3.30 – Création de l'objet ImportData pour la définition des paramètres à fournir au terrain

```
1 Ogre::Terrain::ImportData imp;
2 imp.inputImage = &img;  //recuperation de l'image
3 imp.terrainSize = img.getWidth();  //recuperation de la taille de↩
    l'image
4 imp.worldSize = 8000;  //indique la taille du terrain
5 imp.inputScale = 600;  //echelle adoptee pour l'altitude du ↩
    terrain
6 imp.minBatchSize = 33; //taille min du batch pour le terrain
7 imp.maxBatchSize = 65; //taille max du batch pour le terrain
```

On commence par récupérer l'image et sa taille avec les lignes 2 et 3. **Étant donné que les terrains sont carrés, votre image doit elle aussi être carrée**, faites attention à cela.

Ensuite, le paramètre worldSize indique la taille du terrain, c'est-à-dire la longueur de ses côtés en unités de la scène. Plus ce nombre est grand, plus l'image est agrandie.

inputScale correspond à l'échelle adoptée pour l'altitude du terrain. C'est la hauteur qui sépare un point de la carte représenté par un pixel noir d'un point représenté par un pixel blanc. Il doit donc être choisi en parallèle avec la taille du monde, puisque s'il est trop élevé et que le monde est trop petit, vous aurez un relief très escarpé.

18. Comment ecrire un chemin avec des Slashs ?

Les deux dernières valeurs minBatchSize et maxBatchSize renseignent les tailles minimale et maximale de batch pour notre terrain.

La Batch Size

Le mot anglais batch signifie "lot" ou "paquet". L'affichage de modèle 3D à l'écran consommant beaucoup de ressources, plutôt que de chercher à calculer dans les moindres détails la façon dont apparaît la pelouse à l'autre bout du paysage, pour ensuite ne l'afficher que sur une toute petite surface de l'écran, le moteur va simplifier les choses et calculer de façon grossière l'affichage de ces objets.

Ainsi, les textures peuvent être simplifiées, mais aussi les meshs, dont l'ordinateur va réduire le nombre de vertices pour avoir moins de calculs à faire, vu que vous ne voyez pas les détails (on parle aussi de niveau de détail, ou LOD).

Pour un terrain, le maillage pourrait donc avoir un aspect similaire à celui-ci (image issue du wiki d'Ogre3D.org) : Image utilisateur

Le terrain est divisé en lots dont la taille varie en fonction de la distance de la caméra à ces lots. Plus on s'éloigne, plus le lot est simplifié par suppression de vertices. Lorsque plusieurs lots atteignent une taille minimale, ils sont regroupés en un seul lot, qui est à son tour simplifié progressivement si la caméra continue de reculer.

La zone où se situe la caméra est la plus détaillée, le reste est simplifié.

Si la taille minimum de batch est faible, les lots adjacents auront plus facilement un niveau de détail équivalent, mais il y aura plus de lots à gérer par l'ordinateur. En revanche, si elle est élevée, on regroupe plus rapidement les lots, mais les frontières entre ceux-ci sont plus facilement visibles, car le niveau de détail peut varier plus fortement.

Les valeurs que j'ai mises sont des valeurs courantes, sachez juste que la taille maximum est de 65 et qu'**elles doivent obéir à la formule suivante : taille=2n+1**

Mise en place des textures

Pour gérer les textures, l'outil Terrain d'Ogre utilise des calques. Chacun de ces calques correspond à une texture, que vous pourrez ensuite appliquer où bon vous semblera.

Comme on parle de calques, autant vous dire tout de suite qu'il est possible de les superposer, de donner plus ou moins d'intensité à un calque, pour créer des effets élaborés.

Nous allons commencer avec une seule texture pour faire simple et assimiler le principe. Tout se fait à l'aide de notre importateur de données :

Listing 3.31 – Mise en place d'une texture pour le terrain

```
1 //donne la taille de la liste de calques
2 imp.layerList.resize(1);
3
4 //donne la taille de la texture dans le monde
5 imp.layerList[0].worldSize = 100;
6
7 //les deux lignes suivantes inserent chacune une texture dans ↩
      notre calque
```

```
8 imp.layerList[0].textureNames.push\_back(''grass\_green-01\↩
      _diffusespecular.dds'');
9 imp.layerList[0].textureNames.push\_back(''grass\_green-01\↩
      _normalheight.dds'');
```

Ici les fonctions sont relativement explicites.

La première ligne donne la taille de la liste de calques, ici je n'en ai mis qu'un seul. La seconde ligne donne la taille de la texture dans le monde. **Plus le nombre**[19] **est important, plus la texture sera zoomée, et inversement.**

Les deux lignes suivantes insèrent chacune une texture dans notre calque (textureNames est un vector).

Deux textures ? Je croyais qu'on mettait les textures sur des calques différents ?

En fait, on devrait plutôt dire qu'un calque contient un matériau.

Les matériaux sont faits avec deux textures :

— une texture diffuse, qui contient les couleurs, les motifs du matériau ;

— une texture normale, contenant des informations sur le relief du matériau.

La combinaison de ces deux textures permet d'avoir un matériau complet.

Filtrage anisotrope

Je vais revenir rapidement sur le niveau de détail, qui est réglable pour les matériaux via le filtrage de texture.

Vous pouvez régler la netteté du placage de textures sur vos meshs via un niveau de filtrage.

On peut distinguer quatre options de filtrages, de la plus grossière à la plus précise :

— aucun filtrage ;

— bilinéaire ;

— trilinéaire ;

— anisotrope.

Voici la différence entre un filtrage anisotrope (à gauche) et une texture sans filtrage (à droite). La différence est relativement subtile ici mais visible tout de même.

Image utilisateur

Je vous propose donc d'opter pour un filtrage anisotrope, avec une valeur de 8 (la valeur par défaut est 1 et équivaut à l'absence de filtrage). Vous devez donc rajouter ces deux lignes dans votre code, au début de la méthode createScene() par exemple.

Listing 3.32 – Choix d'un filtrage anisotrope

```
1 Ogre::MaterialManager::getSingleton().setDefaultTextureFiltering(↩
      Ogre::TFO\_ANISOTROPIC);
2 Ogre::MaterialManager::getSingleton().setDefaultAnisotropy(8);
```

Le filtrage de textures n'est pas propre uniquement aux terrains, mais affecte toutes les textures affichées par le moteur.

Chargement et nettoyage

Une fois que les paramètres ont été définis dans l'ImportData, il ne reste qu'à préparer et charger le terrain :

19. ce nombre est-il le nombre affecté à la taille de la texture ?

Listing 3.33 – Préparation et chargement du terrain

```
1 mTerrain->prepare(imp);
2 mTerrain->load();
```

Pour terminer et faire un peu de place en mémoire, il est conseillé d'appeler la méthode suivante qui se chargera de libérer la mémoire allouée temporairement pour la création de votre terrain. Placez donc cette ligne à la fin de la méthode createTerrain().

Listing 3.34 – Libération de place en mémoire

```
1 mTerrain->freeTemporaryResources();
```

Compilez et lancez l'application pour obtenir un joli paysage !

3.17 Le plaquage de textures

Ogre nous permet d'utiliser différents calques pour nos matériaux. On va pouvoir mettre les calques les uns sur les autres, modifier leur opacité pour avoir une texture plus ou moins visible, tout en décidant de la zone où l'on veut appliquer la texture.

La première étape consiste à rajouter des calques dans notre liste, avant de créer le terrain. Remplacez le bloc que vous aviez par le code suivant, afin d'ajouter deux nouveaux matériaux.

Listing 3.35 – Ajout de calques

```
1  imp.layerList.resize(3);
2
3  imp.layerList[0].worldSize = 100;
4  imp.layerList[0].textureNames.push_back("grass_green-01↩
       _diffusespecular.dds");
5  imp.layerList[0].textureNames.push_back("grass_green-01↩
       _normalheight.dds");
6
7  imp.layerList[1].worldSize = 30;
8  imp.layerList[1].textureNames.push_back("growth_weirdfungus-03↩
       _diffusespecular.dds");
9  imp.layerList[1].textureNames.push_back("growth_weirdfungus-03↩
       _normalheight.dds");
10
11 imp.layerList[2].worldSize = 200;
12 imp.layerList[2].textureNames.push_back("↩
       dirt_grayrocky_diffusespecular.dds");
13 imp.layerList[2].textureNames.push_back("↩
       dirt_grayrocky_normalheight.dds");
14
15 mTerrain->prepare(imp);
16 mTerrain->load();
```

Par défaut, c'est uniquement le premier matériau qui est affiché donc si vous exécutez le code maintenant, rien n'aura changé sur votre terrain. Il est donc nécessaire d'ajouter quelques lignes pour dire de quelle façon nous voulons faire notre plaquage de texture.

Le principe est simple à comprendre, chaque calque d'un terrain possède un objet TerrainLayerBlendMap (que j'appellerai dorénavant Blend Map) qui

exprime la façon dont le calque est fusionné avec les calques inférieurs (le calque le plus bas est le calque 0).

Cette fusion est simplement une affaire de transparence. Les calques sont placés les uns sur les autres, et la composante transparente indique si la texture en dessous est plus ou moins visible. Il est de plus possible de faire varier la transparence du calque en chaque point de celui-ci, ce qui permet d'avoir un placage par zone.

À la suite du bloc de code précédent, commencez par récupérer les Blend Map correspondant au calque numéro 1, que nous venons d'ajouter (nous nous occuperons du second ensuite).

Listing 3.36 – Récupération du Blend Map pour le premier terrain

```
1  Ogre::TerrainLayerBlendMap* blendMap1 = mTerrain->↩
       getLayerBlendMap(1);
```

Nous allons commencer par plaquer une seule texture au-dessus de l'herbe, sur toute la surface de notre terrain.

L'idée est de parcourir l'ensemble des points du calque avec deux boucles for (il y a autant de points qu'il y avait de pixels dans notre heightmap) et de leur attribuer la transparence désirée, entre 1 (totalement opaque) et 255 (transparent).

La valeur la plus faible est bien 1 et non 0. Si vous mettez 0, la texture est transparente !

Listing 3.37 – Attribution de la transparence désirée sur tous les points du calque

```
1  float* pBlend1 = blendMap1->getBlendPointer();
2
3  for (Ogre::uint16 y = 0; y < mTerrain->getLayerBlendMapSize(); ++↩
       y)
4  {
5      for (Ogre::uint16 x = 0; x < mTerrain->getLayerBlendMapSize()↩
           ; ++x)
6      {
7          // opacite desiree pour le point courant
8          *pBlend1++ = 150;
9      }
10 }
```

Pour terminer, il faut mettre à jour notre Blend Map en appelant les méthodes dirty() puis update(). La première sert à préciser que les données de la Blend map sont obsolètes et doivent être mises à jour, tandis que la seconde fait effectivement la mise à jour.

Si vous n'appelez pas d'abord dirty(), update() n'aura aucun effet.

Listing 3.38 – Mise à jour de la Blend Map

```
1  blendMap1->dirty();
2  blendMap1->update();
```

Je vous mets le code complet de la méthode createTerrain() si vous voulez vérifier que tout est en ordre :

Listing 3.39 – createTerrain (code complet)

```
1  mTerrain = OGRE_NEW Ogre::Terrain(mSceneMgr);
```

```
2
3  // options globales
4  mGlobals = OGRE_NEW Ogre::TerrainGlobalOptions();
5  mGlobals->setMaxPixelError(10);
6  mGlobals->setCompositeMapDistance(8000);
7  mGlobals->setLightMapDirection(mLight->getDerivedDirection());
8  Globals->setCompositeMapAmbient(mSceneMgr->getAmbientLight());
9  mGlobals->setCompositeMapDiffuse(mLight->getDiffuseColour());
10 Ogre::Image img;
11 img.load("terrain.png", Ogre::ResourceGroupManager::↩
       DEFAULT_RESOURCE_GROUP_NAME);
12
13 // informations g\'eom\'etriques
14 Ogre::Terrain::ImportData imp;
15 imp.inputImage = &img;
16 imp.terrainSize = img.getWidth();
17 imp.worldSize = 8000;
18 imp.inputScale = 600;
19 imp.minBatchSize = 33;
20 imp.maxBatchSize = 65;
21
22 // textures
23 imp.layerList.resize(3);
24 imp.layerList[0].worldSize = 100;
25 imp.layerList[0].textureNames.push_back("grass_green-01↩
       _diffusespecular.dds");
26 imp.layerList[0].textureNames.push_back("grass_green-01↩
       _normalheight.dds");
27 imp.layerList[1].worldSize = 30;
28 imp.layerList[1].textureNames.push_back("growth_weirdfungus-03↩
       _diffusespecular.dds");
29 imp.layerList[1].textureNames.push_back("growth_weirdfungus-03↩
       _normalheight.dds");
30 imp.layerList[2].worldSize = 200;
31 imp.layerList[2].textureNames.push_back("↩
       dirt_grayrocky_diffusespecular.dds");
32 imp.layerList[2].textureNames.push_back("↩
       dirt_grayrocky_normalheight.dds");
33 mTerrain->prepare(imp);35mTerrain->load();
34
35 // plaquage de texture
36 Ogre::TerrainLayerBlendMap* blendMap1 = mTerrain->↩
       getLayerBlendMap(1);
37 float* pBlend1 = blendMap1->getBlendPointer();
38 for (Ogre::uint16 y = 0; y < mTerrain->getLayerBlendMapSize(); ++↩
       y)
39 {
40     for (Ogre::uint16 x = 0; x < mTerrain->getLayerBlendMapSize()↩
           ; ++x)
41     {
42         *pBlend1++ = 150;
43     }
44 }
45
46 blendMap1->dirty();
47 blendMap1->update();
48
49 mTerrain->freeTemporaryResources();
```

Vous pouvez maintenant exécuter l'application ! Profitez-en pour vérifier en vous approchant du sol que l'on distingue bien l'herbe et par-dessus la terre. Image utilisateur

D'accord, la forme n'est pas terrible. Mais le principe y est, c'est un début.

On va tout de suite essayer de faire quelque chose de plus esthétique.

Pour cela, maintenant que vous avez saisi les grandes étapes, je vous propose un mini-TP pour vous entraîner.

3.17.1 Au boulot !

Objectif

Une image vaut sûrement mieux qu'un long discours, voici donc ce que vous allez devoir obtenir : Image utilisateur

L'idée est de texturer le terrain en fonction de l'altitude. Si l'on est en dessous d'un certain seuil, il n'y a que de la terre qui apparaît ; au-dessus, on a de l'herbe.

Indications

Je vous donne tout de même les méthodes qui sont utiles, notamment pour trouver l'altitude du terrain en fonction de la position :

Listing 3.40 – méthode getHeightAtTerrainPosition pour trouver l'altitude du terrain en fonction de la position

```
1 float Ogre::Terrain::getHeightAtTerrainPosition(Ogre::Real x, ↩
    Ogre::Real y)
```

Cette fonction retourne l'altitude en fonction de la position, sachant que x et y sont compris entre 0 et 1.

Pour récupérer ces deux valeurs, on utilise une méthode de TerrainLayerBlendMap qui convertit les coordonnées de l'image en coordonnées du terrain (celles dont vous avez besoin) :

Listing 3.41 – Méthode convertImageToTerrainSpace pour convertir les coordonnées de l'image en coordonnées du terrain

```
1 void Ogre::TerrainLayerBlendMap::convertImageToTerrainSpace(↩
    size_t x, size_t y, Ogre::Real * outX, Ogre::Real * outY)
```

Enfin, je vous laisse choisir l'altitude limite entre l'herbe et la terre. Vous avez toutes les cartes en mains maintenant.

Correction

Comme avez dû le deviner, tout se passe dans les deux boucles for.

Pour chaque point parcouru, on recherche ses coordonnées dans le repère du terrain, puis on récupère la hauteur, que l'on compare à notre hauteur limite. Si on est en dessous, on affiche la texture de terre avec une opacité maximum, sinon on ne fait qu'incrémenter le pointeur de la Blend map.

Voici donc le code modifié :

Listing 3.42 – Attribution de la transparence désirée sur tous les points du calque selon leur position

```
1 for (Ogre::uint16 y = 0; y < mTerrain->getLayerBlendMapSize(); ++↩
    y)
2 {
```

```
3       for (Ogre::uint16 x = 0; x < mTerrain->getLayerBlendMapSize()-
            ; ++x)
4       {
5           Ogre::Real terrainX, terrainY;
6           blendMap1->convertImageToTerrainSpace(x, y, &terrainX, &-
                terrainY);
7           Ogre::Real height = mTerrain->getHeightAtTerrainPosition(-
                terrainX, terrainY);
8           if(height < 200)
9               *pBlend1 = 1;
10          pBlend1++;
11      }
12  }
```

Vous pouvez aussi bien sûr récupérer la Blend map numéro 2 plus haut pour voir le résultat avec une autre texture, c'est ce que j'ai fait pour obtenir l'image référence pour le TP.

3.17.2 Pour aller plus loin

Sur le même principe, nous allons voir comment appliquer deux textures différentes sur une petite largeur, à une altitude donnée.

Ceci peut être utile si vous voulez réaliser des étendues d'eau dans votre terrain : au bord de l'eau, il y a de la boue, un peu au-dessus, de la terre sèche, puis ensuite l'herbe reprend ses droits. C'est ce que nous allons faire, avec une opacité progressive, mais sans l'eau, ce sera pour plus tard.

Nous devons commencer par récupérer un pointeur sur notre seconde Blend Map pour pouvoir gérer la seconde texture en plus de la première. Il y a donc deux lignes à rajouter en conséquence avant les boucles.

Listing 3.43 – Récupération des Blend Map pour le premier et le second terrain

```
1   Ogre::TerrainLayerBlendMap* blendMap1 = mTerrain->-
        getLayerBlendMap(1);
2   Ogre::TerrainLayerBlendMap* blendMap2 = mTerrain->-
        getLayerBlendMap(2);
3
4   float* pBlend1 = blendMap1->getBlendPointer();
5   float* pBlend2 = blendMap2->getBlendPointer();
```

Définissons aussi deux variables pour chacune des textures : la hauteur à laquelle se situe la texture et la largeur de la bande que l'on veut obtenir.

```
1   Ogre::Real minHeight1 = 70;
2   Ogre::Real fadeDist1 = 40;
3   Ogre::Real minHeight2 = 70;
4   Ogre::Real fadeDist2 = 15;
```

Dans la boucle, on déclare trois variables : deux coordonnées du terrain et la transparence pour le point actuel.

```
1   Ogre::Real terrainX, terrainY, transparence;
```

On récupère ensuite la hauteur du terrain comme précédemment.

```
1   blendMap1->convertImageToTerrainSpace(x, y, &terrainX, &terrainY)-
        ;
2   Ogre::Real height = mTerrain->getHeightAtTerrainPosition(terrainX-
        , terrainY);
```

Ensuite, pour chaque texture, on calcule la différence entre la hauteur du point actuel et la hauteur que l'on veut pour la texture, divisée par la largeur de la bande. Si le point est censé être recouvert par la texture, ce nombre sera donc compris entre 0 et 1.

On utilise ensuite la méthode statique Clamp() qui a pour prototype :

Listing 3.44 – Utilisation de la méthode statique Clamp()

```
1 static T Ogre::Math::Clamp(T val, T minval, T maxval)
```

Si val est inférieure à minval, la fonction retourne minval ; si val est supérieure à maxval, on retourne maxval. Si val est dans l'intervalle, on la retourne directement.

Comme on ne veut afficher que les points dont la valeur calculée précédemment est comprise entre 0 et 1, on va utiliser cette méthode pour "couper" toutes les valeurs en dehors de l'intervalle.

```
1 transparence = (height - minHeight1) / fadeDist1;
2 transparence = Ogre::Math::Clamp(transparence, (Ogre::Real)0, (↩
    Ogre::Real)1);
```

Pour terminer, on multiplie transparence par 255 pour avoir une valeur comprise entre 0 et 255.

```
1 *pBlend1++ = transparence * 255;
```

On observe que si transparence est à l'extérieur de l'intervalle [0 ; 1] après le premier calcul, Clamp retournera 0 ou 1. Quand on multiplie par 255, on obtient donc 0 ou 255, qui sont les deux valeurs pour lesquelles la texture est transparente. Mission accomplie !

On copie ces trois lignes pour la seconde texture, et on obtient le code suivant dans nos boucles :

```
1  for (Ogre::uint16 y = 0; y < mTerrain->getLayerBlendMapSize(); ++↩
       y)
2  {
3      for (Ogre::uint16 x = 0; x < mTerrain->getLayerBlendMapSize()↩
           ; ++x)
4      {
5          Ogre::Real terrainX, terrainY, transparence;
6          blendMap1->convertImageToTerrainSpace(x, y, &terrainX, &↩
               terrainY);
7          Ogre::Real height = mTerrain->getHeightAtTerrainPosition(↩
               terrainX, terrainY);
8          transparence = (height - minHeight1) / fadeDist1;
9          transparence = Ogre::Math::Clamp(transparence, (Ogre::↩
               Real)0, (Ogre::Real)1);
10         *pBlend1++ = transparence * 255;
11         transparence = (height - minHeight2) / fadeDist2;
12         transparence = Ogre::Math::Clamp(transparence, (Ogre::↩
               Real)0, (Ogre::Real)1);
13         *pBlend2++ = transparence * 255;
14     }
15 }
```

Pensez à mettre à jour la seconde Blend Map une fois que les modifications sont terminées :

```
1 blendMap1->dirty();
2 blendMap2->dirty();
3 blendMap1->update();
4 blendMap2->update();
```

3.18 Les groupes de terrains

Un groupe de terrains (ou TerrainGroup) range les terrains comme dans un tableau à deux dimensions. Dans le groupe, tous les terrains doivent avoir la même taille, afin de pouvoir les aligner dans une grille.

3.18.1 Création

On commence par ajouter un include au début de la classe puis on remplace notre instance de Terrain par un TerrainGroup, ensuite on l'initialisera dans notre méthode createTerrain() :

Listing 3.45 – TerrainGroup : include et création

```
1 #include <Ogre/Terrain/OgreTerrainGroup.h>
2 //...
3 Ogre::TerrainGroup *mTerrainGroup;
```

Après les lignes permettant de charger l'image heightmap dans la méthode createTerrain(), insérez les lignes suivantes.

```
1 mTerrainGroup = OGRE_NEW Ogre::TerrainGroup(mSceneMgr, Ogre::↩
    Terrain::ALIGN_X_Z, img.getWidth(), 8000);
2
3 //On definit ensuite la position de l'origine du groupe de ↩
    terrains
4 mTerrainGroup->setOrigin(Ogre::Vector3::ZERO);
5
6 //nom (et l'extension) que l'on veut attribuer a nos fichiers qui↩
     seront crees pour sauvegarder les terrains
7 mTerrainGroup->setFilenameConvention(Ogre::String("TerrainDuZero"↩
    ), Ogre::String("dat"));
```

Les paramètres à fournir au TerrainGroup sont

- le Scene manager,
- l'alignement du terrain par rapport au repère global (vous pouvez créer un terrain vertical, par exemple),
- la taille des heightmaps utilisées,
- la taille d'un terrain.

On définit ensuite la position de l'origine du groupe de terrains, puis le nom (et l'extension) que l'on veut attribuer à nos fichiers qui seront créés pour sauvegarder les terrains par la suite.

Pour les données des terrains enregistrées dans un objet ImportData, nous allons modifier un peu le fonctionnement du programme. On récupère en fait directement une référence sur un ImportData fourni par le TerrainGroup, que l'on modifie directement et qui sera valable pour l'ensemble des terrains du groupe. À noter que la ligne de définition de la heightmap n'est plus utile ici, cela sera indiqué lors de la création des terrains.

```
1  Ogre::Terrain::ImportData& imp = mTerrainGroup->↩
       getDefaultImportSettings();
2  imp.terrainSize = img.getWidth();
3  imp.worldSize = 8000;
4  imp.inputScale = 600;
5  imp.minBatchSize = 33;
6  imp.maxBatchSize = 65;
```

Il est maintenant temps de créer les terrains du groupe. On définit la taille du groupe et pour chaque case, on appelle une méthode definirTerrain() définie plus bas qui s'occupera de créer chaque terrain indépendamment.

Listing 3.46 – Création des terrains du groupe

```
1  int largeur = 2, longueur = 2;
2
3  for(int x = 0; x < largeur; x++)
4  {
5      for(int y = 0; y < longueur; y++)
6      {
7          definirTerrain(x, y);
8      }
9  }
10 mTerrainGroup->loadAllTerrains(true);
```

Le groupe se charge pour terminer d'appeler les méthodes load() de chaque terrain à travers la méthode loadAllTerrains(). Cette méthode prend un booléen en paramètre qui indique si le chargement doit être synchrone, c'est-à-dire exécuté dans un seul thread (le thread principal ici). Par défaut cette valeur est fausse, c'est-à-dire que les terrains sont chargés dans plusieurs threads si c'est possible.

Le chargement devient vite très lourd si l'on ajoute beaucoup de terrains aux groupes. Nous verrons plus bas comment accélérer le chargement.

Maintenant, nous devons écrire la méthode definirTerrain() qui utilise la méthode defineTerrain() de TerrainGroup. Celle-ci va prendre 3 paramètres : les deux coordonnées du terrain dans le groupe de terrains (sa position sur la grille, donc) et l'image heightmap utilisée pour ce terrain. Les coordonnées du terrain au sein du groupe peuvent être négatives.

Juste avant d'appeler le terrain, on va faire une vérification sur les coordonnées : si l'abscisse du terrain est impaire, on inverse l'image suivant l'axe Y, si l'ordonnée est impaire, on inverse l'image cette fois-ci selon l'axe X. Cela permet aux terrains du groupe de ne pas avoir de différence d'altitude lors des jointures. Si vous utilisez des heightmaps différents sur les terrains du groupe (ce qui sera probablement le cas), vous devrez faire attention à ce que les altitudes des bords correspondent pour éviter les trous à ces endroits.

Listing 3.47 – PremiereApplication.definirTerrain

```
1  void PremiereApplication::definirTerrain(int x, int y)
2  {
3      Ogre::Image img;
4      img.load("terrain.png", Ogre::ResourceGroupManager::↩
           DEFAULT_RESOURCE_GROUP_NAME);
5
6      if(x % 2 != 0)
7          img.flipAroundY();
8
```

```
9       if(y % 2 != 0)
10          img.flipAroundX();
11
12      mTerrainGroup->defineTerrain(x, y, &img);
13  }
```

Une fois que les terrains sont chargés, il faut leur appliquer les textures définies. On utilise pour cela un itérateur sur le groupe de terrains et, pour chaque terrain, on appelle une méthode initBlendMaps() qui contient le code pour texturer les terrains.

```
1   Ogre::TerrainGroup::TerrainIterator ti = mTerrainGroup->↩
        getTerrainIterator();
2
3   while(ti.hasMoreElements())
4   {
5       Ogre::Terrain* t = ti.getNext()->instance;
6       initBlendMaps(t);
7   }
```

La méthode initBlendMaps() contient uniquement du code que l'on a déjà vu mais que j'ai déplacé pour plus de clarté. Elle prend en paramètre le terrain dont on doit modifier les Blend maps.

```
1   void PremiereApplication::initBlendMaps(Ogre::Terrain *terrain)
2   {
3       Ogre::TerrainLayerBlendMap* blendMap1 = terrain->↩
            getLayerBlendMap(1);
4       Ogre::TerrainLayerBlendMap* blendMap2 = terrain->↩
            getLayerBlendMap(2);
5       Ogre::Real minHeight1 = 70;
6       Ogre::Real fadeDist1 = 40;
7       Ogre::Real minHeight2 = 70;
8       Ogre::Real fadeDist2 = 15;
9       float* pBlend1 = blendMap1->getBlendPointer();
10      float* pBlend2 = blendMap2->getBlendPointer();
11
12      for (Ogre::uint16 y = 0; y < terrain->getLayerBlendMapSize();↩
            ++y)
13      {
14          for (Ogre::uint16 x = 0; x < terrain->↩
                getLayerBlendMapSize(); ++x)
15          {
16              Ogre::Real terrainX, terrainY, transparence;
17              blendMap1->convertImageToTerrainSpace(x, y, &terrainX↩
                    , &terrainY);
18              Ogre::Real height = terrain->↩
                    getHeightAtTerrainPosition(terrainX, terrainY);
19              transparence = (height - minHeight1) / fadeDist1;
20              transparence = Ogre::Math::Clamp(transparence, (Ogre↩
                    ::Real)0, (Ogre::Real)1);
21              *pBlend1++ = transparence * 255;
22              transparence = (height - minHeight2) / fadeDist2;
23              transparence = Ogre::Math::Clamp(transparence, (Ogre↩
                    ::Real)0, (Ogre::Real)1);
24              *pBlend2++ = transparence * 255;
25          }
26      }
27      blendMap1->dirty();
28      blendMap2->dirty();
29      blendMap1->update();
30      blendMap2->update();
```

```
31 }
```

Pour terminer, comme avec un terrain seul, on libère la mémoire utilisée par le TerrainGroup.

```
1 mTerrainGroup->freeTemporaryResources();
```

Votre scène doit maintenant avoir une surface plus grande que la premiÃ¨re fois (on a maintenant quatre terrains). Vous pouvez encore augmenter le nombre de terrains, mais attention, le temps de chargement augmente rapidement !

3.18.2 Optimiser le temps de chargement

Vous avez certainement remarqué que la génération du terrain prend un temps conséquent lorsque le groupe s'agrandit. La création du terrain à partir du fichier heightmap nécessite en effet de convertir les données de l'image en données exploitables par le moteur.

Afin de réduire le temps de chargement, il est possible d'enregistrer un fichier qui contient toutes les informations sur le terrain construit pour éviter de relire l'image à chaque lancement de l'application. L'inconvénient majeur est la place occupée par ces fichiers générés, qui contiennent beaucoup plus d'informations qu'une simple heightmap.

En regardant la création du groupe de terrains, vous voyez que l'on a défini une convention de nommage pour des fichiers, mais qui est pour l'instant inutilisée.

```
1 mTerrainGroup->setFilenameConvention(Ogre::String("TerrainDuZero"↩
    ), Ogre::String("dat"));
```

Cette ligne sert lors de la sauvegarde de fichiers de terrain : ceux-ci seront nommés en commenÃ§ant par Â« TerrainDuZero Â» suivi d'un nombre permettant d'identifier le terrain, puis de l'extension de fichier Â« dat Â».

Pour utiliser la sauvegarde des terrains, nous allons ajouter un attribut mTerrainCreated à la classe PremiereApplication qui permettra de savoir si l'on a généré le terrain à partir d'une image ou bien si l'on a lu un fichier terrain. Dans le premier cas, on saura qu'à la fin de la méthode createTerrain() il faut penser à sauvegarder les fichiers de terrain pour le prochain lancement de l'application.

```
1 bool mTerrainCreated;
```

Initialisez sa valeur à false au début de la méthode createTerrain().

Maintenant, dans notre méthode definirTerrain(), il faut vérifier si le fichier terrain existe déjà ou bien s'il faut faire la génération depuis le heightmap comme le faisait jusqu'alors. Dans le second cas, on passe la variable mTerrainCreated à true.

```
1 void PremiereApplication::definirTerrain(int x, int y)
2 {
3     if(Ogre::ResourceGroupManager::getSingleton().resourceExists(↩
          mTerrainGroup->getResourceGroup(), mTerrainGroup->↩
          generateFilename(x, y)))
4     {
5         mTerrainGroup->defineTerrain(x, y);
6     }
7     else
```

```
8       {
9           Ogre::Image img;
10          img.load("terrain.png", Ogre::ResourceGroupManager::↩
                DEFAULT_RESOURCE_GROUP_NAME);
11
12          if(x % 2 != 0)
13              img.flipAroundY();
14
15          if(y % 2 != 0)
16              img.flipAroundX();
17
18          mTerrainGroup->defineTerrain(x, y, &img);
19          mTerrainCreated = true;
20      }
21  }
```

Revenons sur la condition à tester pour vérifier l'existence du fichier généré.

Grâce au Ogre : :ResourceGroupManager, on peut vérifier s'il existe une ressource précise dans l'ensemble des ressources chargées au démarrage du programme. Les paramètres de la méthode sont le groupe de ressources dans lequel on veut chercher la ressource ainsi que le nom du fichier recherché. Vous voyez que le nom du fichier généré par le groupe de terrains dépend de ses coordonnées X et Y, ainsi que de la convention que l'on a définie au début.

Si le fichier est trouvé, on appelle la méthode definieTerrain() avec seulement les coordonnées en paramètres. Dans ce cas, Ogre va aller chercher directement le fichier correspondant à ces coordonnées. Dans le cas contraire, on exécute le bloc que l'on avait précédemment et qui charge le terrain à partir de l'image de heightmap.

Il ne reste plus qu'à demander la sauvegarde des fichiers si l'on a généré les terrains juste avant de libérer les ressources dans la méthode createTerrain() :

```
1   if(mTerrainCreated)
2       mTerrainGroup->saveAllTerrains(true);
```

Lancez l'application, le temps de chargement doit être un peu plus long qu'auparavant car l'ordinateur sauvegarde en même temps les fichiers générés sur le disque. Une fois que l'application est lancée, fermez-la puis relancez-la. Le temps de chargement doit normalement être meilleur.

Vous devriez trouver les fichiers générés dans le dossier OgreSDK.media. Pour information, les miens font chacun une taille de 12 Mo.

Annexe A

LaTeX

A.1 Vérifier sa configuration latex

Lors d'une importation de ce document sur un autre pc que celui utilisé pour écrire ce document des problèmes peuvent apparaître lors de la compilation de ce fichier tex.

Pour vérifier que aucun package latex ne manque créer un fichier tex et y copier/coller le code suivant. Ce code reprend tous les packages utilisés pour l'écriture de ce présent document (prise de notes á partir du tutorial du site des zéro sur Ogre)

Listing A.1 – Code Latex minimal pour tester les packages latex nécessaires á la compilation du fichier ****-ergo.tex par texmaker

```
1
2  \documentclass[10pt,a4paper]{report}
3  \usepackage[latin1]{inputenc}
4  \usepackage{amsmath}
5  \usepackage{amsfonts}
6  \usepackage{amssymb}
7  \usepackage{hyperref} %pr inserer des liens internet
8
9  \usepackage{verbatim}%pour linsertion brute de commande LaTeX ↩
       dans le texte
10 \usepackage{moreverb}
11
12 \usepackage[french]{babel}
13 \usepackage[latin1]{inputenc}
14
15 \usepackage{listings}
16 \usepackage{xcolor}
17
18 \usepackage{makeidx}
19
20 \title{OGRE}
21 \author{O}
22
23 \begin{document}
24 This is a MINIMUM WORKING EXAMPLE. hgf\newline
25
26 \'e
```

```
\‘e
\^e
\newline

\’A
\‘A
\^S
\end{document}
```

A.2 Caractères accentués

Pour faire un ù

`\‘{u}`

Pour faire un â

`\^{a}`

Pour faire un à

`\‘{a} ou \‘a`

Pour faire un ö

`\’’{o} ou \"o`

Pour faire un î

`\^{i}`

Pour faire un è

`\‘e`

Pour faire un ê

`\^e`

Pour faire un é

`\’{e} ou \’e`

Pour faire un ç

`\c{c}`

Pour faire un ȯ

`\.{o} ou \.o`

Pour faire un ũ

`\~{u} ou \~u`

Pour faire un ō

`\={o} ou \=o`

Pour faire un û

`\^u`

Pour faire des guillemets

`’’guillemets’’`

A.2.1 Méthode alternative (non testée)

Il semble possible d'insérer directement tous les caractères français, pour démo tester le code suivant :

Listing A.2 – Insertion de caractères français

```
1 \documentclass[10pt,a4paper]{article}
2 \usepackage[utf8]{inputenc}
3 \usepackage[T1]{fontenc}
4 \begin{document}
5 entrer des caracteres accentues francais
6 guillemets
7 \end{document}
```

Le problème est que celà a des conséquences sur le code déjà écrit.

A.3 Notes

A.3.1 Note dans la marge

Une note dans la marge

ceci est une note dans la marge

A.3.2 Note de bas de marge

Une note de bas de page[1].

A.4 Bloc commenté

Un bloc commenté se fait avec le package verbatim

```
\begin{comment}
bloc comment\'e
\end{comment}
```

A.5 Numérotation des chapitres et autres

Apparemment le fait d'écrire

```
\section
```

fait que la section sera numérotée,

```
\section*
```

ne sera pas numérotée.

1. Comme celle-ci.

A.6 Exemple d'insertion de code avec lstlisting

L'insertion de code se fait grace aux packages :

```
\usepackage{listings} %pr inserer du code
\usepackage{xcolor}
```

On peut ensuite définir certain paramètres d'insertion

Listing A.3 – Commande pour spécifier les paramètres d'insertion de code

```
\lstset{
basicstyle=\small, % print whole listing small
keywordstyle=\color{blue}\bfseries\underbar,
% underlined bold black keywords
identifierstyle=, % nothing happens
commentstyle=\color{white}, % white comments
stringstyle=\ttfamily, % typewriter type for strings
showstringspaces=false} % no special string spaces
```

```
for i:=maxint to 0 do
begin
{ do nothing }
end;
Write('Case insensitive ');
Write('Pascal keywords.');
```

cf `http://tex.stackexchange.com/questions/21106/adding-c-code-in-latex`

A.7 Créer un index

Pour créer un index : cf `http://www.tuteurs.ens.fr/logiciels/latex/makeindex.html`

pour que l'index soit géré par TexMaker : cf `http://www.xm1math.net/doculatex/makeindex.html`

A.8 Créer une annexe

Pour crér une annexe il faut utiliser la commande

Listing A.4 – Créer une annexe

```
\appendix
\part{Test}     % une "Annexe A Test" sera creee
\chapter{test}  % sera creee "A.1 test"
```